C# Collections

A Detailed Presentation

```
/*****************************************************************
 *  RBInsertFixUp Method
 * ***************************************************************/
private void RBInsertFixUp(Node<TKey, TValue> node) {
  while ((node.Parent != null) && (node.Parent.IsRed)) {
    Node<TKey, TValue> y = null;
    if ((node.Parent.Parent != null) && (node.Parent == node.Parent.Parent.Left)) {
      y = node.Parent.Parent.Right;

      if ((y != null) && (y.IsRed)) {   //case 1
        node.Parent.MakeBlack();        //case 1
        y.MakeBlack();                  //case 1
        node.Parent.Parent.MakeRed();   //case 1
        node = node.Parent.Parent;

        if (node.IsRed) {
          continue;
        }

      } else if (node == node.Parent.Right) { //case 2
        node = node.Parent;                   //case 2
        RotateLeft(node);                     //case 2
      }

      /**************/
      if ((node.Parent != null)) {            //case 3
        node.Parent.MakeBlack();              //case 3
        if (node.Parent.Parent != null) {     //case 3
          node.Parent.Parent.MakeRed();       //case 3
          RotateRight(node.Parent.Parent);    //case 3
        }
      }
      /*******************/

    } else {                                //Parent is a right child

      if (node.Parent.Parent != null) {
        y = node.Parent.Parent.Left;
      }

      if ((y != null) && (y.IsRed)) {       //case 1
        node.Parent.MakeBlack();            //case 1
```

Rick Miller

C# Collections

A Detailed Presentation

C# Collections

A Detailed Presentation

Rick Miller

Pulp Free Press
Falls Church, Virginia

Pulp Free Press, Falls Church, Virginia 22042
www.pulpfreepress.com
info@pulpfreepress.com

First edition published 2012

Pulp Free Press offers discounts on this book when ordered in bulk quantities. For more information regarding sales contact *sales@pulpfreepress.com.*

The PDF edition of this book may be licensed for bulk distribution. For whole or partial content licensing information contact *licensing@pulpfreepress.com*.

Publisher Cataloging-in-Publication Data: *Prepared by Pulp Free Press*

```
Miller, Rick, 1961 -
C# Collections: A Detailed Presentation/Rick Miller

p. cm.
Includes bibliographical references and index.
ISBN-13: 978-1-932504-09-5 ISBN-10: 1-932504-09-5 (pbk)
1. C#(computer program language)  II. Title.
2. Object-Oriented Programming (Computer Science)
QA76.73.C154 M555 2012
005.133M555COL2012     Library of Congress Control Number: 2009903762
                                                                    CIP
```


C# Collections: A Detailed Presentation was meticulously crafted on a Macintosh PowerMac G5 using Adobe FrameMaker, Adobe Illustrator, Macromedia Freehand, Adobe Photoshop, Adobe Acrobat, and Microsoft Word. C# source code examples were prepared using TextPad, NotePad++, and Microsoft Visual Studio. Photographs appearing at the beginning of each chapter were made with a variety of cameras and film as noted in the vertical captions.

Printed in the United States of America.

ISBN-10: 1-932504-09-5; ISBN-13: 978-1-932504-09-5 — First Paperback Edition

To Coralie —
Always wonderful...
Forever beautiful...
Tu es incroyable!
— Rick

Preface

Welcome – And Thank You!

Welcome to *C# Collections: A Detailed Presentation*. Thank you for supporting my writing efforts.

I know your time is extremely valuable and I sincerely appreciate you spending some of it reading this book. A friend of mine likes to say, after wasting time reading a particularly useless reference, "...I'd like that part of my life back!" I can only hope that's not the way you feel when finally you lay down this book. If it is, please don't hesitate to share with me your concerns and ideas for improving this work. You can contact me directly via email at rick@pulpfreepress.com.

Target Audience

This book targets the intermediate to advanced programmer who wants a deeper understanding of the .NET collections framework. I make no attempt to explain fundamental programming concepts nor do I dwell on the fundamentals of C# programming. If you need to brush up on the basics please consult my earlier book: *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming* (ISBN-13: 978-1-932504-07-1)

SupportSite™ Website

The *C# Collections: A Detailed Presentation* SupportSite™ is located at [http://pulpfreepress.com/content/SupportSites/CSharpCollectionsBook/]. The support site includes the complete source code listings by chapter and in one big zip file and an errata with corrections and updates to the text.

Problem Reporting

Although I made every possible effort to produce a work of superior quality, some mistakes will no doubt go undetected. All typos, misspellings, inconsistencies or other problems found in *C# Collections: A Detailed Presentation* are mine and mine alone. To report a problem or issue with the text please contact me directly at rick@pulpfreepress.com or report the problem via the book's SupportSite™. I will happily acknowledge your assistance in the improvement of this book both online and in subsequent editions.

About The Author

Presently, I'm a senior computer scientist and web applications architect for Science Applications International Corporation (SAIC) where I design and build enterprise web applications for the Department of Defense intelligence community. I hold a master's degree in computer science from California State University, Long Beach and am an assistant professor at Northern Virginia Community College, Annandale Campus, where I teach a variety of computer

programming courses. I enjoy reading, writing, and photography. You can view a small sample of my photos at www.warrenworks.com.

Acknowledgments

A lot of really great people are responsible, either directly or indirectly, for making this book a reality. I'd like to collectively thank all my students who've reviewed pieces and parts of the text and helped make significant improvements.

I especially want to thank my friends and colleagues Pete Luongo, Alex Remily, and Tri Nguyen. These guys help keep both work and life interesting.

Lastly, I offer a special thanks to my wife and friend Coralie. She allows me to roam about the cabin and pursue my interests. We go together like peanut butter and jelly!!!

Rick Miller
Falls Church, Virginia

Table of Contents

7 Stacks

12 Sets

13 Thread Programming

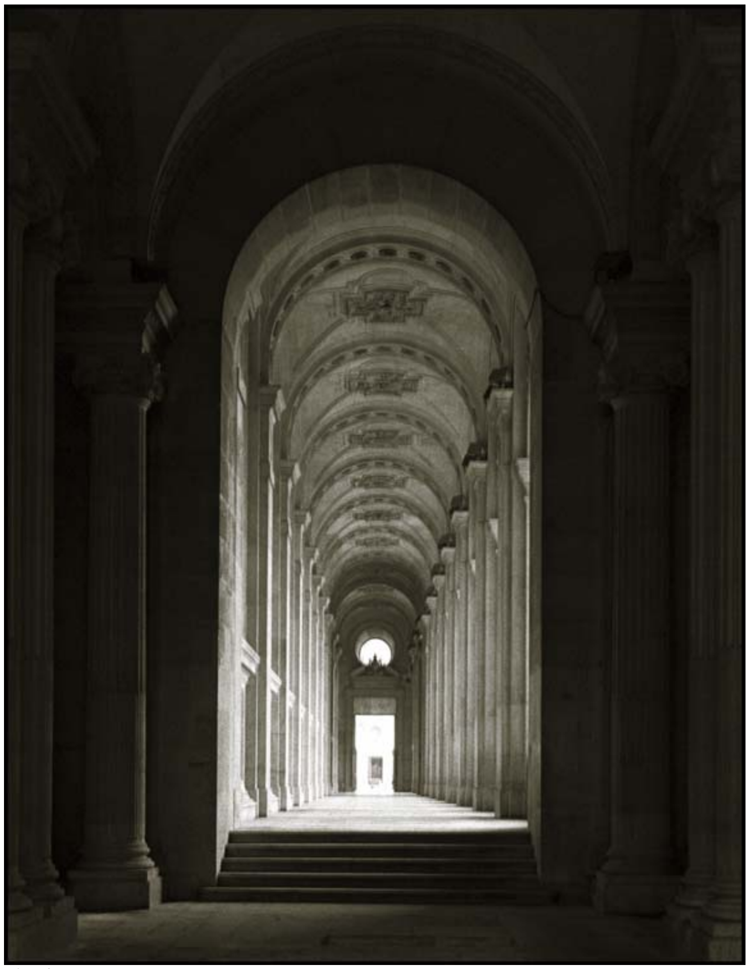
20 Collections In Action

Appendices

List of Figures

List of Tables

Chapter 1



Contax T — Kodak Tri-X Professional

Archway

Collections Quick Start

Learning Objectives

- *Quickly put collection classes to work in your programs*
- *Create and utilize an ArrayList collection*
- *Create and utilize a List<T> collection*
- *Add elements to an ArrayList and List<T> collection*
- *Access elements in an ArrayList and List<T> collection*
- *Cast objects to their proper type upon retrieval from an ArrayList collection*

Introduction

This chapter, as its name implies, provides you with a “quick start” introduction to the .NET collections framework. My intent is to show you how to put collection classes to work immediately solving real programming problems that require the manipulation of a collection of objects.

I will focus the discussion on the use of the ArrayList and List<T> collection classes. Together, these two classes easily represent the most frequently used collection types. Along the way I will explain the difference between early, non-generic .NET collection classes, which in this book I will refer to as “old school” collection classes, and the more recent (as of .NET 2.0) generic collection classes. I still believe it is a good idea to discuss the old school collections because you may find yourself maintaining legacy .NET code.

You will be surprised at how productive you can be by simply arming yourself with the knowledge of how to use these two classes. However, as you progress through this book, you will quickly learn that to gain full advantage of the .NET collection framework you must cultivate a deep understanding of special coding techniques, especially when you want to manipulate collections of your very own user-defined data types.

Why Use a Collection?

The short answer to this question is, “Because it will save you time, and lots of it!” The .NET collections framework, comprised of classes, structures, and interfaces found in four namespaces: *System.Collections*, *System.Collections.Generic*, *System.Collections.ObjectModel*, and *System.Collections.Specialized*, makes your programming life seriously easy by providing a ready-made set of collection types that offer a solution to just about any collection requirement you will encounter.

Before you can tap its full potential you will need to invest some time up front studying the collections framework. You’ll need to understand how one type of collection differs from another, how the underlying data structure used to implement a particular collection affects its performance, and how to code your custom user-defined data types to behave well in a collection. I discuss these and many other topics in detail throughout the book. In the end, you will be generously rewarded for your effort.

However, in the meantime, to demonstrate the power of collections I will show you how to use two popular collection types: ArrayList and List<T>. You would select either of these collection types in situations where you would ordinarily use an array, but wanted more specialized behavior than a simple array offered. Both the ArrayList and List<T> collection classes can be manipulated like an array, and in fact, both collections are array based, so if you already know how to manipulate an array, you already know how to use part of the interface to both of these collection classes. I’ll start the discussion with the old school ArrayList class.

ArrayList Class

The ArrayList class is a “non-generic” collection that behaves like an array on steroids. By non-generic I mean it has been around since the early days of the .NET framework and allows you to insert any type of object into it, and retrieve only System.Object type objects from it, which must be cast to the appropriate type before calling any type-specific interface methods on the retrieved object. This “object in, object out” behavior characterizes the early .NET collections framework.

Another feature of the ArrayList class is its ability to dynamically grow or expand when necessary to accommodate more objects. An ordinary array does not possess this capability. To grow an ordinary array, you must copy its elements to a temporary array, create a bigger permanent array, and then copy the elements from the temporary array back to the new larger permanent array. (Whew!) All this is done automagically for you with an ArrayList collection.

Example 1.1 shows an ArrayList in action manipulating a collection of Strings.

1.1 ArrayListDemo.cs

```
1	using System;
2	using System.Collections;
3
4	public class ArrayListDemo {
5	  public static void Main(){
6	    ArrayList list = new ArrayList();
7
8	    //Add elements using the Add() method
9	    list.Add("Hope Mesa");
10	    list.Add("Bill Hicks");
11	    list.Add("Secret Miller");
12	    list.Add("Alex Remily");
13	    list.Add("Pete Luongo");
14
15	    //access elements using array indexer notation
16	    for(int i=0; i<list.Count; i++){
17	      Console.WriteLine(list[i]);
18	    }
19
20	    Console.WriteLine("------------------------------");
21
22	    //or, use the foreach statement which hides the complexity of the enumerator
23	    foreach(string s in list){
24	      Console.WriteLine(s);
25	    }
26	  }
27	}
```

Referring to example 1.1 — note that to use the ArrayList class you must add the `using` directive as is shown on line 2 to provide shortcut naming access to the members of the System.Collections namespace. On line 6 an ArrayList object is created and assigned to the reference named `list`. Lines 9 through 13 show string objects being added to the list. The `for` statement beginning on line 16 iterates over the list of strings and prints their values to the console. The `foreach` statement that starts on line 23 shows an alternative way to iterate over the collection with the help of an *iterator*. However, the `foreach` statement hides the complexity of the iterator so there's not much to see from looking at the code. Figure 1-1 shows the results of running this program.

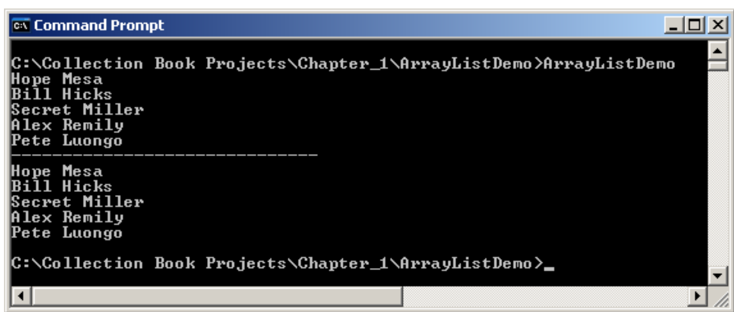


Figure 1-1: Results of Running Example 1.1

In example 1.1, the ArrayList contains only strings, but there's nothing stopping you, except your good sense, from adding any kind of object to the list. To demonstrate, I'll create a custom data type named Dog, the code for which appears in example 1.2.

1.2 Dog.cs

```
public class Dog {
  private string _first_name;
  private string _last_name;
  private string _breed;

  public Dog(string breed, string f_name, string l_name){
    _breed = breed;
    _first_name = f_name;
    _last_name = l_name;
  }

  public string FirstName {
    get { return _first_name; }
    set { _first_name = value; }
  }

  public string LastName {
    get { return _last_name; }
    set { _last_name = value; }
```

```
    }

    public string Breed {
      get {  return _breed; }
      set {  _breed = value; }
    }

    public string FullName {
      get {  return FirstName + " " + LastName; }
    }

    public string BreedAndFullName {
      get {  return Breed + ": " + FullName; }
    }
  }
```

Referring to example 1.2 — the Dog class contains five public properties: FirstName, LastName, Breed, FullName, and BreedAndFullName. The first three are read-write properties and the last two are read-only. Example 1.3 shows an ArrayList being used to store various types of objects: string, integer, and Dog.

1.3 ArrayListDemo.cs (mod 1)

```
1     using System;
2     using System.Collections;
3
4     public class ArrayListDemo {
5       public static void Main(){
6         ArrayList list = new ArrayList();
7
8         //Add various types of objects to the ArrayList
9         list.Add("Baba Beaton");
10        list.Add(1);
11        list.Add(new Dog("Boxer", "Sammy", "Socks"));
12
13        //Access each object in the collection and print out its value
14        foreach(object o in list){
15          Console.WriteLine(o);
16        }
17      }
18    }
```

Referring to example 1.3 — notice now on lines 9 through 11 that I'm adding three different types of objects to the list. The `foreach` statement on line 14 iterates over the list and prints the value of each element to the console. But what value will be printed for the Dog object? Figure 1-2 shows the results of running this program.

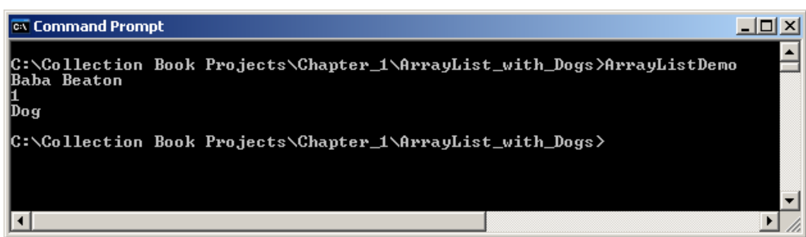


Figure 1-2: Results of Running Example 1.3

Referring to figure 1-2 — the reason this codes works at all is because when the value of each object is printed to the console, its ToString() method is called automatically. This is what happens when supplying an object as a argument to the overloaded Console.WriteLine() method.

When using user-defined types where the Object.ToString() method has **not** been overridden, the default behavior results in the class type being returned, which is what printed to the console in the case of the Dog object. If you want another, more meaningful, value to be printed to console instead of the type name, you must override the Object.ToString() method in the Dog class as example 1.4 illustrates.

1.4 Dog.cs (with overridden ToString() method)

```
public class Dog {
  private string _first_name;
  private string _last_name;
  private string _breed;

  public Dog(string breed, string f_name, string l_name){
    _breed = breed;
    _first_name = f_name;
    _last_name = l_name;
  }

```



```
12        public string FirstName {
13          get {  return _first_name; }
14          set {  _first_name = value; }
15        }
16
17        public string LastName {
18          get {  return _last_name; }
19          set {  _last_name = value; }
20        }
21
22        public string Breed {
23          get {  return _breed; }
24          set {  _breed = value; }
25        }
26
27        public string FullName {
28          get {  return FirstName + " " + LastName; }
29        }
30
31        public string BreedAndFullName {
32          get {  return Breed + ": " + FullName; }
33        }
34
35        //override System.Object.ToString() method
36        public override string ToString(){
37          return BreedAndFullName;
38        }
39      }
```

Referring to example 1.4 — the ToString() method has been overridden on line 36 and returns the BreedAndFullName string. Figure 1-3 shows the results of running example 1.3 with this modified version of the Dog class.

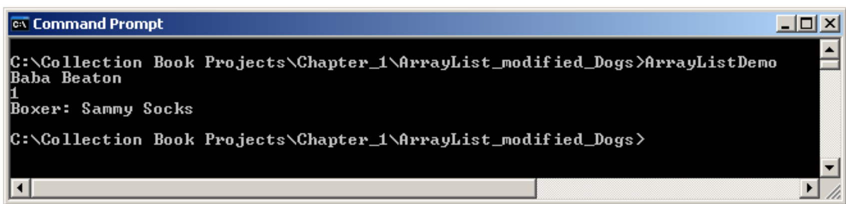


Figure 1-3: Results of Running Example 1.3 with Overridden ToString() Method in the Dog Class

Referring to figure 1-3 — notice now that you get a more meaningful value from a Dog object because of the overridden ToString() method.

Polymorphic Behavior

The previous example illustrated how, when manipulating a collection of objects, polymorphic behavior can be utilized to treat all objects as if they were the same type, usually some targeted base type. In the case of an ArrayList, it can hold any type of object, but all the contained objects, when accessed, are returned as a System.Object. Thus, as long as your code targets the System.Object interface, all the list's contained objects can be treated uniformly or polymorphically.

Casting to a Specific Type

If, however, you want to access a member on an object retrieved from an ArrayList that does not belong to the System.Object's interface, you must attempt to convert that object into the specified type with a casting operation. Example 1.5 shows how a list of Dog objects must be cast to the Dog type before Dog-specific interface properties can be accessed.

1.5 ArrayListCastingDemo.cs

```
using System;
using System.Collections;

public class ArrayListCastingDemo {
  public static void Main(){
    ArrayList list = new ArrayList();
    list.Add(new Dog("Boxer", "Sammy", "Socks"));
    list.Add(new Dog("Golden Retriever", "Woody", "Miller"));
    list.Add(new Dog("Yellow Lab", "Austin", "Miller"));
```



```
10
11          //explicitly cast each retrieved object to the Dog type
12          for(int i=0; i<list.Count; i++){
13            Console.WriteLine(((Dog)list[ i] ).BreedAndFullName);
14          }
15
16          Console.WriteLine("------------------------");
17
18          //the foreach statement does the casting for you...
19          foreach(Dog d in list){
20            Console.WriteLine(d.BreedAndFullName);
21          }
22        }
23      }
```

Referring to example 1.5 — I inserted three Dog objects into the list. The `for` loop on line 12 shows how each object must be explicitly cast to the Dog type upon retrieval from the list before a Dog-specific interface member can be accessed. If you use a `foreach` statement the compiler does the casting for your if you specify the type of objects you want out of the list. For each of these casting operations to succeed, objects actually contained in the list must successfully cast to the specified type, otherwise you'll throw an InvalidCastException.

Quick Review

An ArrayList is a non-generic collection that behaves like an array on steroids. You can insert any type of object into an ArrayList, but upon retrieval, all contained objects are returned as type System.Object. If you want to access type-specific members on a retrieved object, you must cast the retrieved object to that type.

An ArrayList will dynamically grow to accommodate more objects. This dynamic growth is handled automatically. When the number of inserted objects reaches a certain threshold, the list is resized.

List<T> Class

The .NET framework 2.0 introduced generic collections. A generic collection allows you to specify the type of objects you want to insert into the collection and thus eliminates the need to cast upon retrieval. This "specific type in, specific type out" behavior characterizes the generic collection classes.

The List<T> class is the generic replacement for the old-school ArrayList class. Nowadays, you should prefer the use of generics over the use of old-school collections when writing new code.

Example 1.6 demonstrates the use of the List<T> collection class.

1.6 ListTDemo.cs

```
using System;
using System.Collections.Generic;

public class ListTDemo {
  public static void Main(){
    List<String> list = new List<String>();

    //Add elements using the Add() method
    list.Add("Hope Mesa");
    list.Add("Bill Hicks");
    list.Add("Secret Miller");
    list.Add("Alex Remily");
    list.Add("Pete Luongo");

    //access elements using array indexer notation
    for(int i=0; i<list.Count; i++){
      Console.WriteLine(list[ i] .ToUpper());
    }

    Console.WriteLine("------------------------------");

    //or, use the foreach statement which hides the complexity of the enumerator
    foreach(string s in list){
      Console.WriteLine(s);
    }
  }
}
```


Referring to example 1.6 — a List<T> object is declared and created on line 6. The <T> in the angle brackets represents the type placeholder. Replace the T with whatever type you want the list to contain. In this example, I've specified a list of Strings (e.g., List<String>). If you're completely new to generics the syntax takes some getting used to. Note how you must repeat the type specification in the angle brackets when making the constructor call. Once you've created the list, you use it just like an ArrayList. The only exception is that now you know what types of objects the list contains because you specified the type when you created the list. This eliminates the need to cast retrieved objects from the list. The `for` loop on line 16 demonstrates this by calling the ToUpper() method on each string object in the list without casting to the string type. The `foreach` statement remains unchanged from earlier examples.

Quick Review

The generic List<T> class allows you to specify the type of objects the list will contain when you create the list. This eliminates the need to cast retrieved objects.

When writing new code, prefer the use of the generic collection classes over non-generic old-school collections.

Manipulating Lists

There's much more to the ArrayList and List<T> classes than just dynamic growth. Both classes provide a wide assortment of interface methods that allow you to manipulate the collection in many ways. In this section I will show you how to sort the elements of a list, search a list for a specific entry, and reverse the elements of a list. I will use a List<T> collection to demonstrate these operations but you can perform the same operations on an ArrayList.

Sorting, Searching, and Reversing a List<T> Collection

The sorting, searching, and reversing operations, along with many others, are part of the List<T> collection's interface. These three operations take the form of the Sort(), BinarySearch(), and Reverse() methods, all of which are overloaded, meaning there is more than one way to sort, search, and reverse a list.

Default Sorting Behavior

Before you sort a list, you must be aware of what type of objects the list contains. Before two objects can be compared with each other, they must implement the IComparer or IComparer<T> interface or there must exist one or more comparer objects that derive from System.Comparer or System.Comparer<T> that instructs the Sort() method on how to compare each object.

In the case of built-in .NET framework classes, you don't have to worry about such matters. Classes and structures that are meant to be sorted, like strings, integers, characters, etc., already implement the IComparer and IComparer<T> interfaces.

Sort Before Calling The BinarySearch() Method

The title of this section says it all. The BinarySearch() method works on a sorted list, so be sure you have sorted the list before calling the BinarySearch() method.

Example 1.7 demonstrates the use of the Sort(), BinarySearch(), and Reverse() methods on a list of strings.

1.7 ListManipulationDemo.cs

```
using System;
using System.Collections.Generic;

public class ListManipulationDemo {
  public static void Main(){
    List<String> list = new List<String>();

    //Add elements using the Add() method
    list.Add("Hope Mesa");
    list.Add("Bill Hicks");
```



```
11          list.Add("Secret Miller");
12          list.Add("Alex Remily");
13          list.Add("Pete Luongo");
14
15          //Access elements using array indexer notation
16          for(int i=0; i<list.Count; i++){
17            Console.WriteLine(list[ i] );
18          }
19
20          Console.WriteLine("------------------------------");
21
22          //Sort the list using the natural ordering of the String class
23          list.Sort();
24
25          //Print the sorted list to the console
26          foreach(string s in list){
27            Console.WriteLine(s);
28          }
29
30           Console.WriteLine("------------------------------");
31
32           //Search the list for a specific string
33           Console.WriteLine("The string \"Hope Mesa\" is located at index number "
34                              + list.BinarySearch("Hope Mesa") + " in the list.");
35
36           Console.WriteLine("------------------------------");
37
38           //Now, reverse the list
39           list.Reverse();
40
41           //Print the reversed list to the console
42          foreach(string s in list){
43            Console.WriteLine(s);
44          }
45        }
46      }
```

Referring to example 1.7 — a list of strings is declared and created on line 6 and then populated with string objects on lines 9 through 13. The `for` loop on line 16 writes the strings to the console. On line 23, the list is sorted with a call to its Sort() method and the list of sorted strings is then written to the console.

Line 34 demonstrates the use of the BinarySearch() method. The BinarySearcy() method searches the list for the specified object, in this case a string (e.g., "Hope Mesa") and returns its index.

On line 39, the list is reversed with a call to the Reverse() method. The reversed list is then written to the console with the `foreach` statement on line 42.

Figure 1.4 shows the results of running this program.

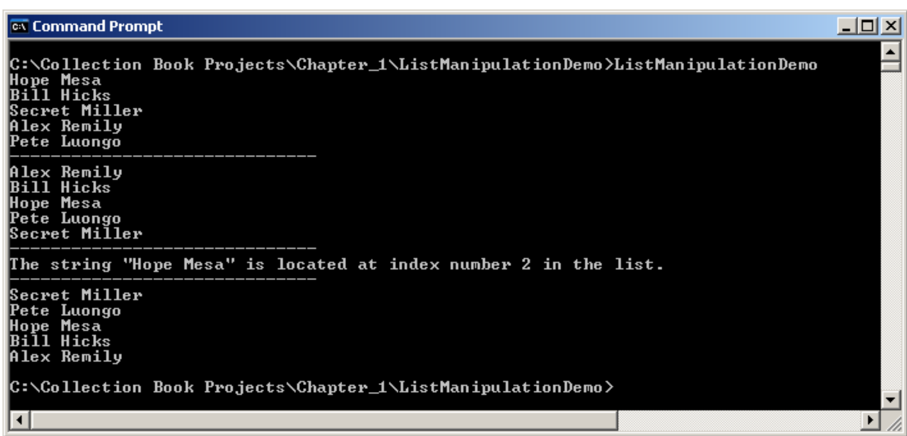


Figure 1-4: Results of Running Example 1.7

Quick Review

The ArrayList and List<T> collection classes provide a wide assortment of methods that let you manipulate the collections in many ways. Three helpful operations include Sort(), BinarySearch(), and Reverse().

Before one object can be compared to another they must be comparable. This means they must implement the IComparable or IComparable<T> interface or one or more comparer objects must exist that extend the Comparer or Comparer<T> classes.

.NET framework classes that are meant to be sorted already implement both the IComparable and IComparable<T> interfaces.

Remember to sort a list before calling the BinarySearch() method.

Where To Go From Here

If you haven't already done so, now would be a good time to explore the .NET framework documentation on the Microsoft Developer Network (MSDN) website (www.msdn.com). Pay particular attention to the members of the four collection namespaces: *System.Collections*, *System.Collections.Generic*, *System.Collections.ObjectModel*, and *System.Collections.Specialized*. Look up the ArrayList and List<T> classes and study their methods and properties.

Summary

An ArrayList is a non-generic collection that behaves like an array on steroids. You can insert any type of object into an ArrayList, but upon retrieval, all contained objects are returned as type System.Object. If you want to access type-specific members on a retrieved object, you must cast the retrieved object to that type.

An ArrayList will dynamically grow to accommodate more objects. This dynamic growth is handled automatically. When the number of inserted objects reaches a certain threshold, the list is resized.

The generic List<T> class allows you to specify the type of objects the list will contain when you create the list. This eliminates the need to cast retrieved objects.

When writing new code, prefer the use of the generic collection classes over non-generic old-school collections.

The ArrayList and List<T> collection classes provide a wide assortment of methods that let you manipulate the collections in many ways. Three helpful operations include Sort(), BinarySearch(), and Reverse().

Before one object can be compared to another they must be comparable. This means they must implement the IComparable or IComparable<T> interface or one or more comparer objects must exist that extend the Comparer or Comparer<T> classes.

.NET framework classes that are meant to be sorted already implement both the IComparable and IComparable<T> interfaces.

Remember to sort a list before calling the BinarySearch() method.

References

.NET Framework 3.5 Reference Documentation, Microsoft Developer Network (MSDN) [www.msdn.com]

Notes

Chapter 2

Contax T

Sidewalk

Collections Framework Overview

Learning Objectives

- *List the four namespaces that constitute the .NET collections framework*
- *Understand the organization and content of the .NET collections framework*
- *List the members of each .NET collection framework namespace*
- *Navigate the .NET framework application programming interface (API) documentation*
- *List the non-generic collection classes and their corresponding generic replacements*

Introduction

As elementary as the material in this chapter may appear upon initial consideration, it pays huge dividends to know your way around the .NET framework documentation. You will spend countless hours studying the documentation no matter how many books you read, because that's where you'll find the most up to date information on the classes, structures, and other components of the .NET application programming interface (API).

In this chapter I show you how to decipher exactly what functionality a particular collection class provides by studying its inheritance hierarchy and list of implemented interfaces. I then highlight the major collection components found in the four namespaces of the .NET collections framework.

I also provide you with a helpful listing of the non-generic collection classes and their generic replacements.

The Microsoft Developer Network Documentation

The first step towards getting good help is knowing where to find it. The .NET API documentation hosted on the Microsoft Developer Network (MSDN) is the definitive source for the latest information regarding the .NET framework.

There are two ways of gaining access to the .NET framework documentation: 1. The straight forward way, which is to go directly to the MSDN website, follow the links to the docs, and then bookmark the link, or 2. The fastest way, which is to enter the name of the class or component you're looking for into Google. This, of course, assumes you know what you're looking for. If you take the time to explore the .NET framework API, you'll have a good idea of what to look for.

The MSDN Website – www.msdn.com

The Microsoft Developer Network is the community portal for developers using Microsoft technology. In addition to many other areas of interest, it hosts the .NET framework API documentation. The URL to the site is [http://www.msdn.com]. Figure 2-1 shows the MSDN homepage at the time of this writing.

Figure 2-1: Microsoft Developer Network (MSDN) Home Page

On the MSDN homepage locate the .NET Framework link in the lower left corner as figure 2-1 illustrates. Click the link. This takes you to the .NET Framework Developer Center page which is shown in figure 2-2.

Home tab is initially selected.

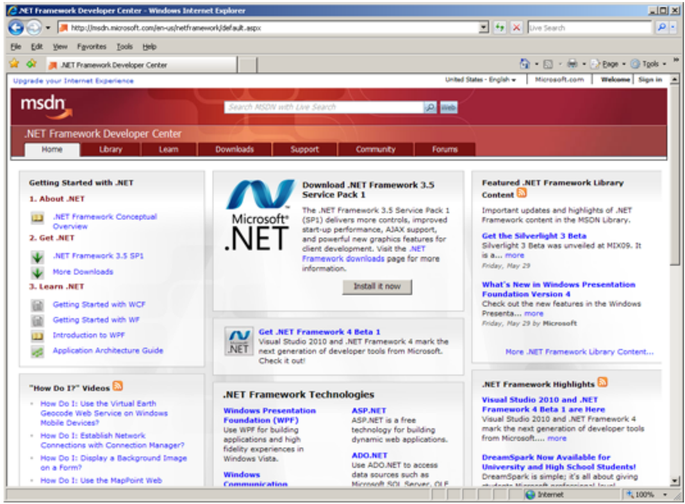


Figure 2-2: .NET Developer Center Home Tab

Referring to figure 2-2 — the Home tab is initially selected when you arrive at the .NET Framework Developer Center page. Click the Library tab to access the documentation as is shown in figure 2-3.

Click the Library tab to access the documentation.

Figure 2-3: .NET Developer Center Library Tab

Next, locate the .NET Development link in the left frame as figure 2-4 illustrates.

Select .NET Development link

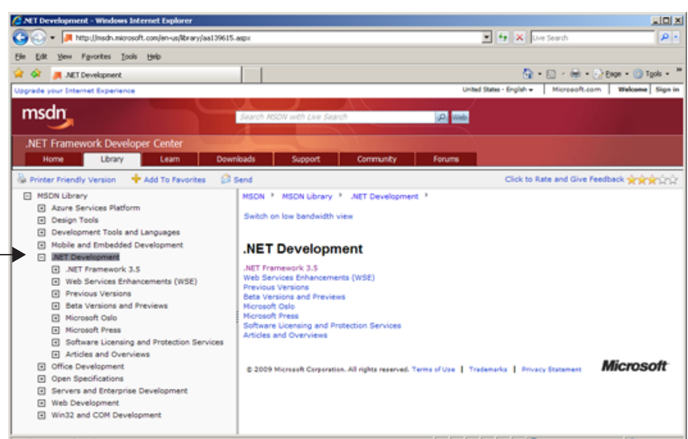


Figure 2-4: .NET Development Link Selected

Underneath the .NET Development link, locate and expand the .NET Framework 3.5 link, then locate the .NET Framework Class Library link as is shown in figure 2-5.

Select .NET Framework Class Library link

Figure 2-5: .NET Framework Class Library Link

Click the .NET Framework Class Library link to expand its contents as shown in figure 2-6. A description of the .NET Framework Class Library appears in the right hand frame. Bookmark the site for future reference.

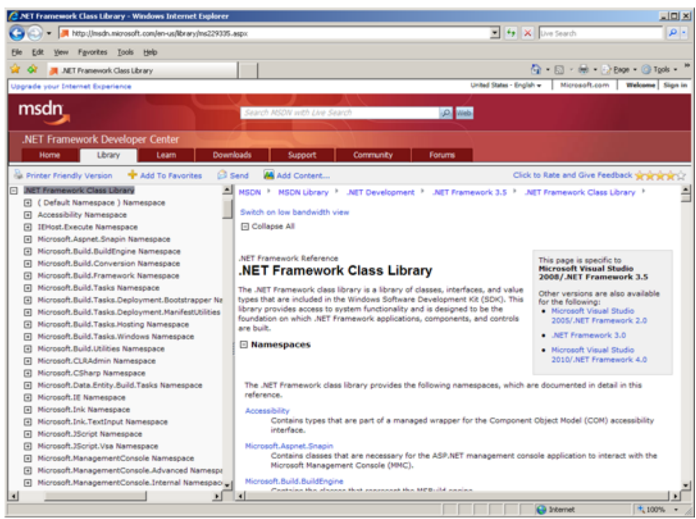


Figure 2-6: .NET Framework Class Library Home

Using Google to Quickly Locate Documentation

A faster way to access the .NET framework documentation is to search for a particular namespace or component in Google or your favorite web browser. This works best if you have some idea of what you're looking for. Figures 2-7 through 2-9 show how Google can be used to search for and quickly locate the System.Collections namespace.

Where to Go From Here

Now that you know where to find .NET framework documentation, I recommend taking the time to explore the .NET framework class library docs to get a feel for what's there. Explore the *System*, *System.Collections*, and *System.Collections.Generic* namespaces. Get a good feel for the classes, structures, and other components located in each namespace. Study their methods and properties. At first this seems like a daunting task, but if you devote a little time each day to studying a small piece of the documentation, you'll quickly learn your way around.

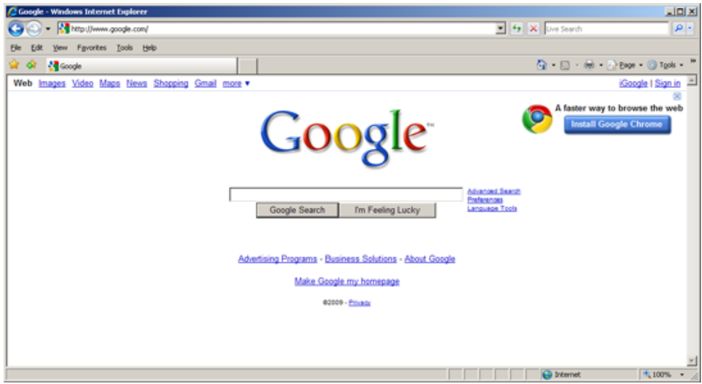


Figure 2-7: Google Home Page

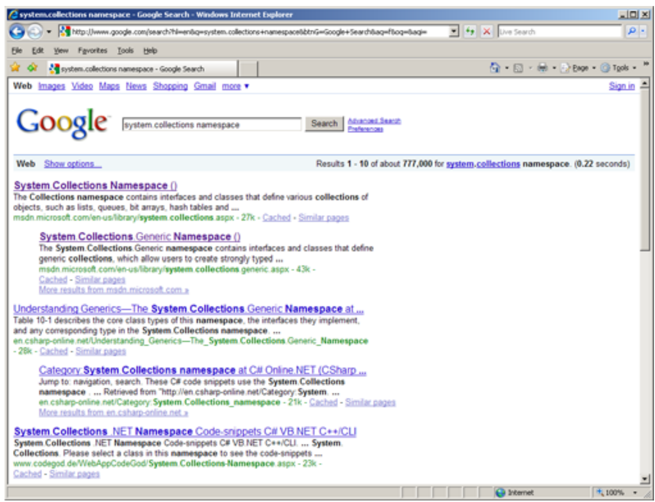


Figure 2-8: Search Results for System.Collections Namespace

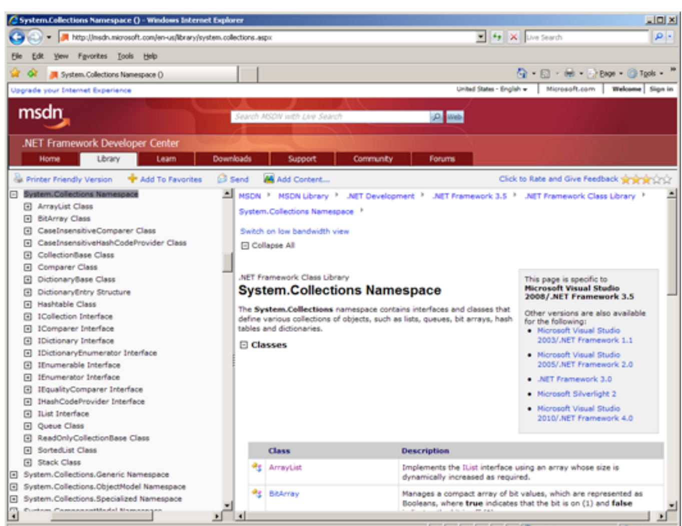


Figure 2-9: System.Collections Namespace Documentation

Quick Review

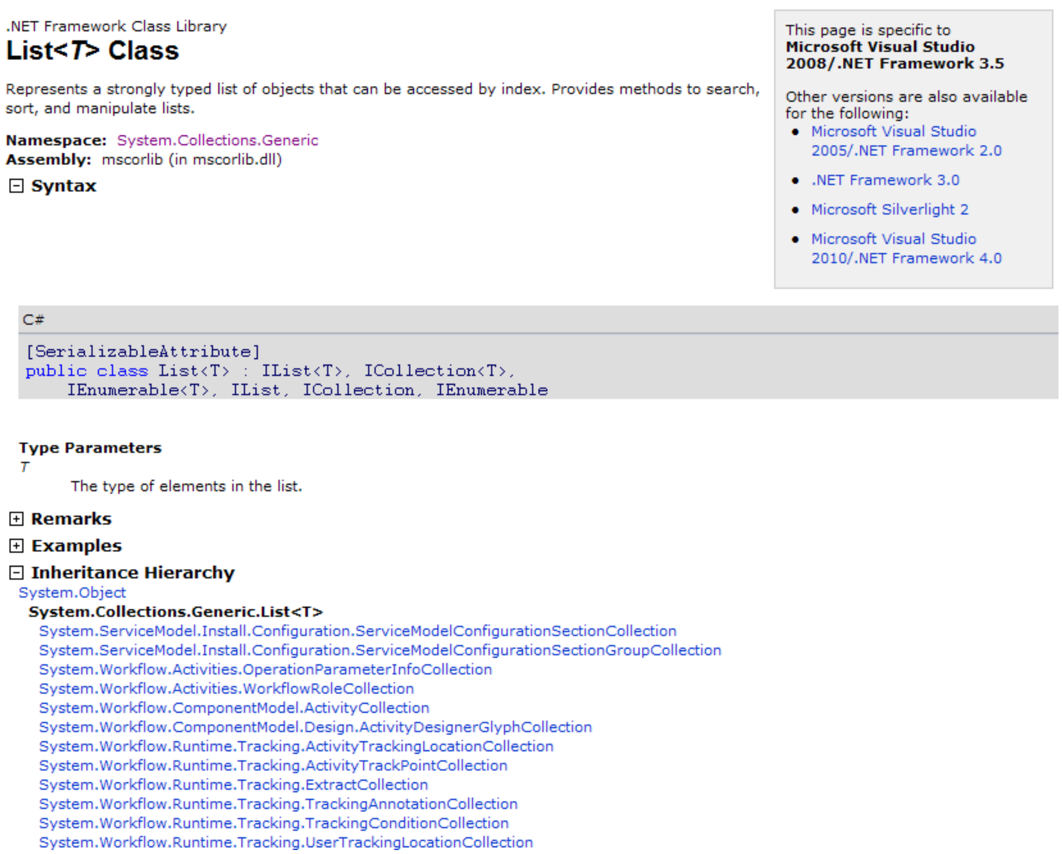
The Microsoft Developer Network (MSDN) contains the latest .NET framework API documentation. It's a good idea to explore the site and bookmark the .NET Framework Class Library page for future reference.

Use Google to quickly locate MSDN API documentation pertaining to individual .NET framework namespaces or components.

Navigating an Inheritance Hierarchy

When you come across a particular class or structure in the .NET framework, chances are the functionality of that component is the result of inheriting from a base class (which itself may inherit from another base class), the implementation of multiple interfaces, or a combination of both. To better understand what, exactly, a component does, you must be able to trace its inheritance hierarchy and/or track down and study its implemented interfaces.

The best place to find this type of information is in the .NET framework documentation for the particular component you're interested in getting to know better. Consider for a moment the List<T> class. If you navigate to the List<T> documentation page on the MSDN website you'll find inheritance and implementation information in two sections: **Syntax** and **Inheritance Hierarchy** as is shown in figure 2-10.



.NET Framework Class Library

List<T> Class

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

Namespace: System.Collections.Generic
Assembly: mscorlib (in mscorlib.dll)

Syntax

This page is specific to **Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

- Microsoft Visual Studio 2005/.NET Framework 2.0
- .NET Framework 3.0
- Microsoft Silverlight 2
- Microsoft Visual Studio 2010/.NET Framework 4.0

C#

```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, IEnumerable
```

Type Parameters

T

The type of elements in the list.

Remarks

Examples

Inheritance Hierarchy

System.Object
System.Collections.Generic.List<T>
System.ServiceModel.Install.Configuration.ServiceModelConfigurationSectionCollection
System.ServiceModel.Install.Configuration.ServiceModelConfigurationSectionGroupCollection
System.Workflow.Activities.OperationParameterInfoCollection
System.Workflow.Activities.WorkflowRoleCollection
System.Workflow.ComponentModel.ActivityCollection
System.Workflow.ComponentModel.Design.ActivityDesignerGlyphCollection
System.Workflow.Runtime.Tracking.ActivityTrackingLocationCollection
System.Workflow.Runtime.Tracking.ActivityTrackPointCollection
System.Workflow.Runtime.Tracking.ExtractCollection
System.Workflow.Runtime.Tracking.TrackingAnnotationCollection
System.Workflow.Runtime.Tracking.TrackingConditionCollection
System.Workflow.Runtime.Tracking.UserTrackingLocationCollection

Figure 2-10: List<T> MSDN Documentation Page - Syntax and Inheritance Hierarchy Sections Expanded

Referring to figure 2-10 — the **Syntax** section gives you the class declaration. The class declaration is important because it lists the class it extends, if any, and all its implemented interfaces. In the .NET framework an interface begins with the letter 'I'. So, by reading the List<T> class declaration you can tell at a glance it does not inherit from a class other than System.Object, but it implements a long list of interfaces.

The **Inheritance Hierarchy** section highlights the class you're currently visiting in black text. Above it in blue text will be a link to its immediate base class. In the case of the List<T> class, its immediate base class is Sys-

tem.Object. (Note: All classes and structures in the .NET framework ultimately and implicitly inherit from System.Object.) Below the class shown in black will be any subclasses derived from that class. As you can see from looking at figure 2-10, the List<T> class serves as the base class for many different specialized collection classes.

Extension Methods

The .NET framework 3.0 introduced *extension methods*. An extension method is a special type of static method that enables a programmer to add a method to an existing class or structure without having to recompile the code associated with that component. An extension method, although static, is called as if it were an instance method.

The .NET API documentation lists the extension methods available to .NET framework classes and structures in the **Extension Methods** section. To see the list of extension methods for a particular class or structure, navigate to the component's Members page. Figure 2-11 shows the Members page for the List<T> generic collections class.

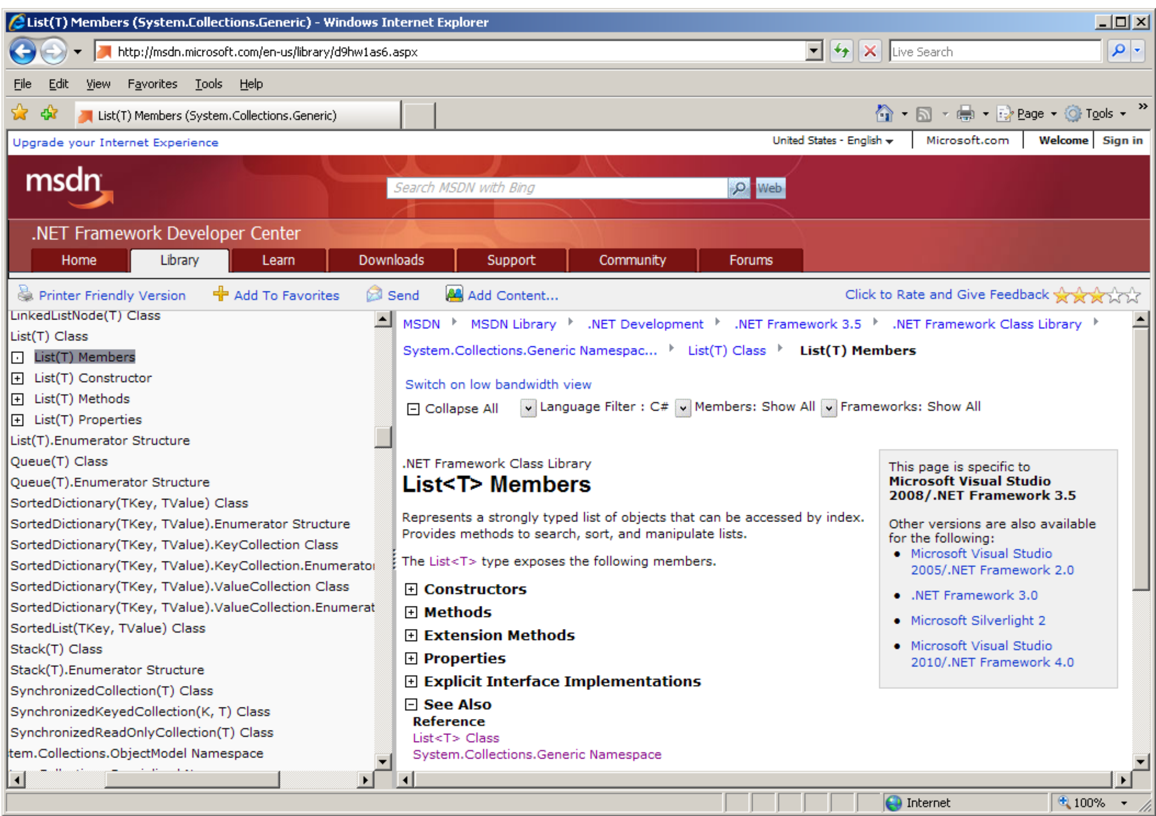


Figure 2-11: Members Page for the List<T> Class

Referring to figure 2-11 — the Members page includes listings for all the constructors, methods, extension methods, properties, and explicit interface implementations. I have collapsed all the sections here to fit the page on the screen.

Constructor methods are defined by the class in question. The sections titled **Methods**, **Properties**, and **Explicit Interface Implementations** list methods and properties either inherited, defined by the class, or implemented by the class to fulfill a contract specified by an interface. The **Extension Methods** section, on the other hand, lists methods that have been added to the class above and beyond those proscribed by inheritance. Most of the methods listed in the List<T>'s **Extension Methods** section are defined by the Enumerable class which is located in the System.Linq namespace.

Quick Review

To get a feel for the functionality a particular class or structure provides, explore the component's inheritance hierarchy, along with its declaration as given in the **Syntax** section of the component's main documentation page. As of .NET framework version 3.0, the **Extension Methods** section lists methods that have been added to the component in addition to those defined by inheritance or interface implementation.

The .NET Collections Framework Namespaces

To the uninitiated, the .NET collections API presents a bewildering assortment of interfaces, classes, and structures spread over four namespaces. In this section I provide an overview of some of the things you'll find in each namespace. Afterward, I present a different type of organization that I believe you'll find more helpful.

One thing I will not do in this section is discuss every interface, class, or structure found in these namespaces. If I did that, you would fall asleep quick and kill yourself as your head slammed against the desk on its way down! Instead, I will only highlight the most important aspects of each namespace with an eye towards saving you time and frustration.

One maddening aspect of the .NET collections framework is in the way Microsoft chose to name their collection classes. For example, collection classes that contain the word List do not necessarily implement the IList or IList<T> interfaces. This means you can't substitute a LinkedList for an ArrayList without breaking your code.

In concert with this section you should explore the collections API and see for yourself what lies within each namespace.

System.Collections

The System.Collections namespace contains non-generic versions of collection interfaces, classes, and structures. The contents of the System.Collections namespace represent the "old-school" way of collections programming. By this I mean that the collections defined here store only object references. You can insert any type of object into a collection like an ArrayList, Stack, etc., but, when you access an element in the collection and want to perform an operation on it specific to a particular type, you must first cast the object to a type that supports the operation. (By "performing an operation" I mean accessing an object member declared by its interface or class type.)

I recommend avoiding the System.Collections namespace altogether in favor of the generic versions of its members found in the System.Collections.Generic namespace. In most cases, you'll be trading cumbersome "old-school" style programming for more elegant code and improved performance offered by the newer collection classes.

System.Collections.Generic

.NET 2.0 brought with it generics and the collection classes found in the System.Collections.Generic namespace. In addition to providing generic versions of the "old-school" collections contained in the System.Collections namespace, the System.Collections.Generic namespace added several new collection types, one of them being LinkedList<T>.

Several collection classes within the System.Collections.Generic namespace can be used off-the-shelf, so to speak, to store and manipulate strongly-typed collections of objects. These include List<T>, LinkedList<T>, Queue<T>, Stack<T>.

Other classes such as Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue>, and SortedList<TKey, TValue> store objects (values) in the collection based on the hash values of keys. Special rules must be followed when implementing a key class. These rules specify the types of interfaces a key class must implement in order to perform equality comparisons. They also offer suggestions regarding the performance of hashing functions to optimize insertion and retrieval. You can find these specialized instructions in the **Remarks** section of a collection class's API documentation page.

System.Collections.ObjectModel

The System.Collections.ObjectModel namespace contains classes that are meant to be used as the base classes for custom, user-defined collections. For example, if you want to create a specialized collection, you can extend the Collection<T> class. This namespace also includes the KeyedCollection<TKey, TItem>, ObservableCollection<T>, ReadOnlyCollection<T>, and ReadOnlyObservableCollection<T> classes.

The KeyedCollection<TKey, TItem> is an abstract class and is a cross between an IList and an IDictionary-based collection in that it is an indexed list of items. Each item in the list can also be accessed with an associated key. Collection elements are not key/value pairs as is the case in a Dictionary, rather, the element is the value and the key is extracted from the value upon insertion. The KeyedCollection<TKey, TItem> class must be extended and you must override its GetKeyForItem() method to properly extract keys from the items you insert into the collection.

The ObservableCollection<T> collection provides an event named CollectionChanged that you can register event handlers with to perform special processing when items are added or removed, or the collection is refreshed.

The ReadOnlyCollection<T> and ReadOnlyObservableCollection<T> classes implement read-only versions of the Collection<T> and ObservableCollection<T> classes.

System.Collections.Specialized

As its name implies, the System.Collections.Specialized namespace contains interfaces, classes, and structures that help you manage specialized types of collections. Some of these include the BitVector32 structure, the ListDictionary, which is a Dictionary implemented as a singly linked list intended for storing ten items or less, StringCollection, which is a collection of strings, and StringDictionary, which is a Dictionary whose key/value pairs are strongly typed to strings rather than objects.

Mapping Non-Generic To Generic Collections

In some cases, the System.Collection.Generic and System.Collections.ObjectModel namespaces provide a corresponding replacement for a collection class in the System.Collections namespace. But sometimes they do not. Table 14-1 lists the non-generic collection classes and their generic replacements, if any, and the underlying data structure implementation.

Non-Generic	Generic	Underlying Data Structure
ArrayList	List<T>	Array
BitArray	*No generic equivalent*	Array
CollectionBase	Collection<T>	Array
DictionaryBase	KeyedCollection<TKey, TItem>	Hash Table & Array
HashTable	Dictionary<TKey, TValue>	Hash Table
Queue	Queue<T>	Array
ReadOnlyCollectionBase	ReadOnlyCollection<T>	Array
SortedList	SortedList<TKey, TValue>	Red-Black Tree
Stack	Stack<T>	Array
No Non-Generic Equivalent	LinkedList<T>	Non-Circular Doubly Linked List
No Non-Generic Equivalent	SortedDictionary<TKey, TValue>	Red-Black Tree
No Non-Generic Equivalent	SynchronizedCollection<T> †	Array

Table 2-1: Mapping Non-Generic Collections to Their Generic Counterparts

Non-Generic	Generic	Underlying Data Structure
No Non-Generic Equivalent	SynchonizedKeyedCollection<TKey, TItem> †	Hash Table & Array
No Non-Generic Equivalent	SynchronizedReadOnlyCollection<T> †	Array
† Provides thread-safe operation		

Table 2-1: Mapping Non-Generic Collections to Their Generic Counterparts

Quick Review

"Old-school" style .NET collection classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval and improved performance. The classes found in the System.Collections.ObjectModel namespace can serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures you will find helpful to manage unique collections.

Summary

The Microsoft Developer Network (MSDN) contains the latest .NET framework API documentation. It's a good idea to explore the site and bookmark the .NET Framework Class Library page for future reference.

Use Google to quickly navigate to documents pertaining to individual .NET namespaces or components.

To get a feel for the functionality a particular class or structure provides, explore the component's inheritance hierarchy, along with its declaration as given in the **Syntax** section of the component's main documentation page. As of .NET framework version 3.0, the **Extension Methods** section lists methods that have been added to the component in addition to those defined by inheritance or interface implementation.

"Old-school" style .NET collections classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval and improved performance. The classes found in the System.Collections.ObjectModel namespace can serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures you will find helpful to manage unique collections.

References

.NET Framework 3.5 Reference Documentation, Microsoft Developer Network (MSDN) [www.msdn.com]

Rick Miller. *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. ISBN-13: 978-1-932404-07-1. Pulp Free Press

Notes

Chapter 3

Contax T

Rain Walkers

Arrays

Learning Objectives

- *Understand the relationship between arrays and collections*
- *Understand the functionality provided by the IList interface*
- *Declare and use single-dimensional arrays*
- *Declare and use multi-dimensional arrays*
- *Understand how arrays are represented in memory*
- *Describe the functionality provided by the System.Array class*
- *Use the System.Array class to sort the contents of an array*
- *Use the System.Array class to reverse the contents of an array*

Introduction

Arrays are closely related to collections in many ways. They serve as the underlying foundational data structure for several collection types including ArrayList, List<T>, and Hashtable, and fill auxiliary roles in others. Arrays also implement the ICollection interface and partially implement the IList interface. Thus, cultivating a thorough working knowledge of arrays leads to a better understanding of collections in general. In fact, it's often desirable to convert a collection onto an array for streamlined manipulation and many collection types provide a ToArray() method for just this purpose.

In this chapter you will learn the meaning of the term *array*, how to create and manipulate single and multidimensional arrays, and how to use arrays in your programs. Starting with single-dimensional arrays of simple predefined value types, you will learn how to declare array references and how to use the `new` operator to dynamically create array objects. To help you better understand the concepts of arrays and their use, I will show you how they are represented in memory. A solid understanding of the memory concepts associated with array allocation helps you to better utilize arrays in your programs. Understanding the concepts and use of single-dimensional arrays enables you to easily understand the concepts behind multidimensional arrays.

Along the way you will learn the difference between arrays of value types and arrays of reference types. I will show you how to dynamically allocate array element objects and how to call methods on objects via array element references. I will also explain to you the difference between *rectangular* and *ragged* arrays.

What Is An Array?

An array is a contiguous memory allocation of same-sized or homogeneous data type elements. *Contiguous* means the array elements are located one after the other in memory. *Same-sized* means that each array element occupies the same amount of memory space. The size of each array element is determined by the type of objects an array is declared to contain. So, for example, if an array is declared to contain integer types, each element would be the size of an integer and occupy 4 bytes. If, however, an array is declared to contain double types, the size of each element would be 8 bytes. The term *homogeneous* is often used in place of the term *same-sized* to refer to objects having the same data type and therefore the same size. Figure 3-1 illustrates these concepts.

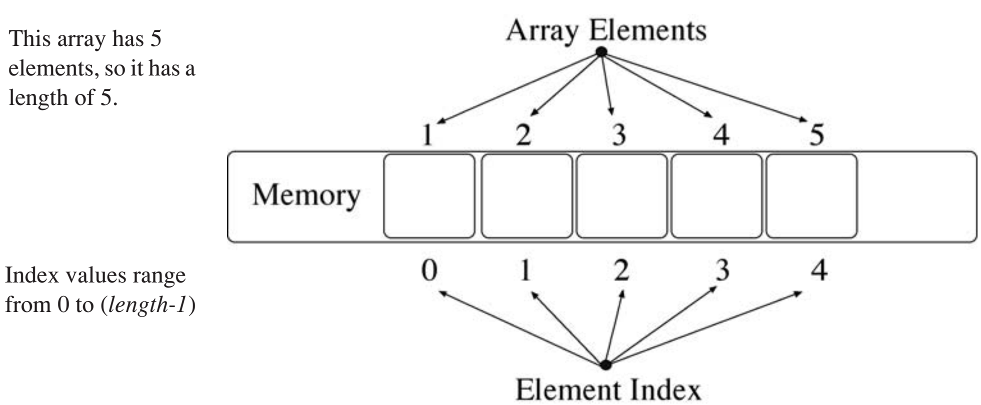


Figure 3-1: Array Elements are Contiguous and Homogeneous

Figure 3-1 shows an array of 5 elements of no specified type. The elements are numbered consecutively, beginning with 1 denoting the first element and 5 denoting the last, or 5th, element in the array. Each array element is referenced or accessed by its array index number. An index number is always one less than the element number it

accesses. For example, when you want to access the 1st element of an array, use index number 0. To access the 2nd element of an array, use index number 1, etc.

The number of elements an array contains is referred to as its *length*. The array shown in figure 3-1 contains 5 elements, so it has a length of 5. The index numbers associated with this array will range from 0 to 4 (that is 0 to *[length - 1]*).

Specifying Array Types

Array elements can be value types, reference types, or arrays of these types. When you declare an array, you must specify the type its elements will contain. Figure 3-2 illustrates this concept through the use of the array declaration and allocation syntax.

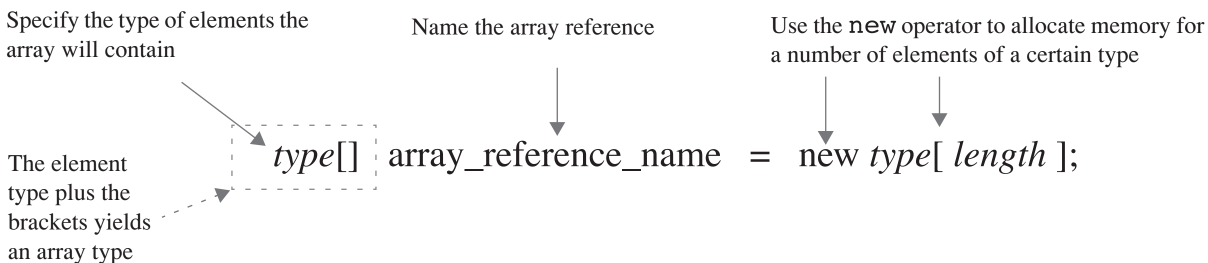


Figure 3-2: Declaring a Single-Dimensional Array

Figure 3-2 shows the array declaration and allocation syntax for a single-dimensional array having a particular *type* and *length*. The declaration begins with the array element type. The elements of an array can be value types or reference types. Reference types can include any reference type specified in the .NET API, reference types you create, or third-party types created by someone else.

The element type is followed by a set of empty brackets. Single-dimensional arrays use one set of brackets. Multidimensional arrays add either commas or brackets, depending on whether you're declaring a rectangular or jagged array. You will add a comma or a set of brackets for each additional *dimension* or *rank* you want a multidimensional array to have. The element type plus the brackets yield an *array type*. This array type is followed by an identifier that declares the name of the array. To actually allocate memory for an array, use the `new` operator followed by the type of elements the array can contain followed by the length of the array in brackets. The `new` operator returns a reference to the newly created array object and the assignment operator assigns it to the array reference name.

Figure 3-2 combines the act of declaring an array and the act of creating an array object on one line of code. If required, you could declare an array in one statement and create the array in another. For example, the following line of code declares and allocates memory for a single-dimensional array of integers having a length of 5:

```
int[] int_array = new int[5];
```

The following line of code would simply declare an array of floats:

```
float[] float_array;
```

And this code would then allocate enough memory to hold 10 float values:

```
float_array = new float[10];
```

The following line of code would declare a two-dimensional rectangular array of boolean-type elements and allocate some memory:

```
bool[,] boolean_array_2d = new bool[10,10];
```

The following line of code would create a single-dimensional array of strings:

```
String[] string_array = new String[8];
```


You will soon learn the details about single and multidimensional arrays. If the preceding concepts seem confusing now just hang in there. By the time you complete this chapter you will be using arrays like a pro!

Quick Review

Arrays are contiguously allocated memory elements of homogeneous data types. Contiguous means the elements are arranged in memory one after the other. Homogeneous means each element of the array is of the same data type. An array containing *n* elements is said to have a length equal to *n*. Access array elements via their index value, which ranges from 0 to (*length - 1*). The index value of a particular array element is always one less than the element number you wish to access (*i.e.,* the 1^{st} element has index 0, the 2^{nd} element has index 1, ... , the n^{th} element has index n-1)

Functionality Provided By C# Array Types

The C# language has two data-type categories: value types and reference types. Arrays are a special case of reference types. When you create an array in C#, it is an object just like a reference type object. However, C# arrays possess special features over and above ordinary reference types because they inherit from the System.Array class. This section explains what it means to be an array type.

Array-Type Inheritance Hierarchy

When you declare an array in C# you specify an array type as was shown previously in figure 3-2. The array you create automatically inherits the functionality provided by the System.Array class which itself extends from the System.Object class. Figure 3-3 shows the UML inheritance diagram for an array type.

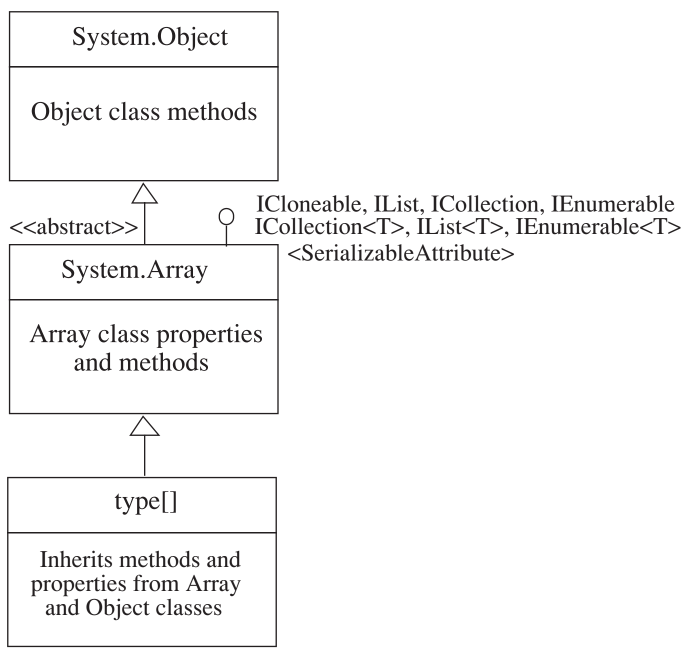


Figure 3-3: Array-Type Inheritance Hierarchy

Referring to figure 3-3 — the inheritance from the Array and Object classes is taken care of automatically by the C# language when you declare an array. The Array class is a special class in the .NET Framework in that you cannot derive from it directly to create a new array type subclass. Any attempt to explicitly extend from System.Array in your code will cause a compiler error.

The Array class provides several public properties and methods that make it easy to manipulate arrays. Some of these properties and methods are non-static and can be accessed via an array reference while others are static and are meant only to be accessed via the Array class itself. You will see examples of the Array class's methods and properties in action as you progress through this chapter. In the meantime, however, it would be a good idea to access the MSDN website and research the System.Array class documentation to learn more about what it has to offer.

Special Properties Of C# Arrays

The Table 3-1 summarizes the special properties of C# arrays.

Property	Description
Their length cannot be changed once created.	Array objects have an associated length when they are created. The length of an array cannot be changed after the array is created. However, arrays can be automatically resized with the help of the Array.Resize() method.

Their number of dimensions or rank can be determined by accessing the Rank property.	For example:
	`int[] int_array = new int[5];`
	This code declares a single-dimensional array of five integers. The following line of code prints to the console the number of dimensions int_array contains:
	`Console.WriteLine(int_array.Rank);`

The length of a particular array dimension or rank can be determined via the GetLength() method.	Array objects have a method named GetLength() that returns the value of the length of a particular array dimension or rank. To call the GetLength() method, use the dot operator and the name of the array. For example:
	`int[] int_array = new int[5];`
	This code declares and initializes an array of integer elements with length 5. The next line of code prints the length of the int_array to the console:
	`Console.WriteLine(int_array.GetLength(0));`
	The GetLength() method is called with an integer argument indicating the desired dimension. In the case of a single-dimensional array, there is only one dimension.

Array bounds are checked by the virtual execution system at runtime.	Any attempt to access elements of an array beyond its declared length will result in a runtime exception. This prevents mysterious data corruption bugs that can manifest themselves when misusing arrays in other languages like C or C++.
Array types directly subclass the System.Array class.	Because arrays subclass System.Array they have the functionality of an Array.
Elements are initialized to default values.	Predefined simple value type array elements are initialized to the default value of the particular value type each element is declared to contain. For example, integer array elements are initialized to zero. Each element of an array of references is initialized to null.

Table 3-1: C# Array Properties

Quick Review

C# array types have special functionality because of their special inheritance hierarchy. C# array types directly and automatically inherit the functionality of the System.Array class and implement the ICloneable, IList, ICollection, and IEnumerable interfaces. Arrays are also serializable.

Creating And Using Single-Dimensional Arrays

This section shows you how to declare, create, and use single-dimensional arrays of both value types and reference types. Once you know how a single-dimensional array works you can easily apply the concepts to multidimensional arrays.

Arrays Of Value Types

The elements of a value type array can be any of the C# predefined value types or value types that you declare (*i.e.*, structures). The predefined value types include *bool*, *byte*, *sbyte*, *char*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *float*, *double*, and *decimal*. Example 3.1 shows an array of integers being declared, created, and utilized in a short program. Figure 3-4 shows the results of running this program.

3.1 IntArrayTest.cs

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        for(int i=0; i<int_array.GetLength(0); i++){
7          Console.Write(int_array[i] + " ");
8         }
9         Console.WriteLine();
10     }
11   }
```

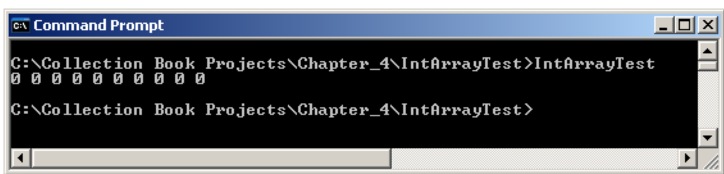


Figure 3-4: Results of Running Example 3.1

Referring to example 3.1 — this program demonstrates several important concepts. First, an array of integers of length 10 is declared and created on line 5. The name of the array is int_array. To demonstrate that each element of the array is automatically initialized to zero, the `for` statement on line 6 iterates over each element of the array beginning with the first element [0] and proceeding to the last element [9], and prints each element value to the console. As you can see from looking at figure 3-4, this results in all zeros being printed to the console.

Notice how each element of int_array is accessed via an index value that appears between square brackets appended to the name of the array (*i.e.*, int_array[i]). In this example, the value of i is controlled by the `for` loop.

How Value-Type Array Objects Are Arranged In Memory

Figure 3-5 shows how the integer array int_array declared and created in example 3.1 is represented in memory. The name of the array, int_array, is a reference to an object in memory of type *System.Int32[]*. The array object is dynamically allocated on the application's memory heap with the `new` operator. Its memory location is assigned to the int_array reference. At the time of array object creation, each element is initialized to the default value for integers which is 0. The array object's Length property returns the value of the total number of elements in the array, which in this case is 10. The array object's Rank property returns the total number of dimensions in the array, which in this case is 1.

Let's make a few changes to the code given in example 3.1 by assigning some values to the int_array elements. Example 3.2 adds another `for` loop to the program that initializes each element of int_array to the value of the `for` loop's index variable i.

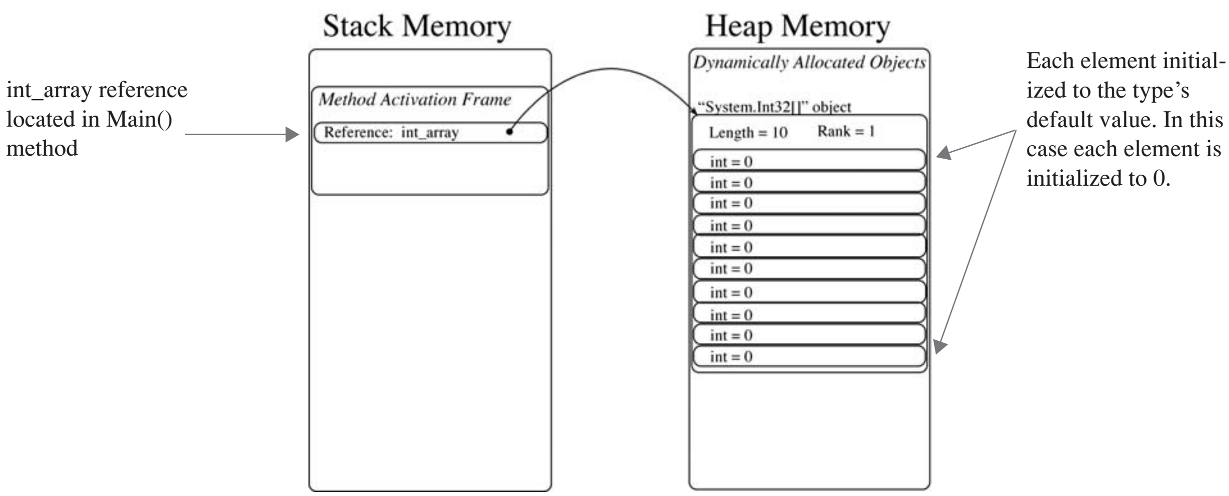


Figure 3-5: Memory Representation of Value Type Array int_array Showing Default Initialization

3.2 IntArrayTest.cs (Mod 1)

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        for(int i=0; i<int_array.GetLength(0); i++){
7          Console.Write(int_array[i] + " ");
8         }
9        Console.WriteLine();
10       for(int i=0; i<int_array.GetLength(0); i++){
11         int_array[i] = i;
12         Console.Write(int_array[i] + " ");
13        }
14       Console.WriteLine();
15     }
16   }
```

Referring to example 3.2 — notice on line 11 how the value of the second `for` loop's index variable i is assigned directly to each array element. When the array elements print to the console, each element's value has changed except for the first, which is still zero. Figure 3-6 shows the results of running this program. Figure 3-7 shows the memory representation of int_array after its elements have been assigned their new values.

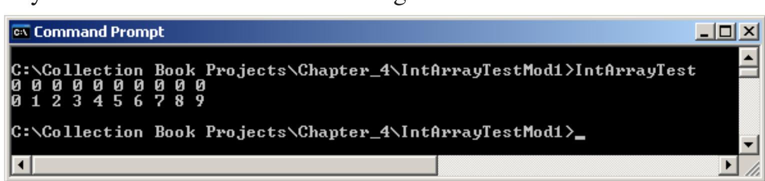


Figure 3-6: Results of Running Example 3.2

Finding An Array's Type, Rank, And Total Number of Elements

Study the code shown in example 3.3, paying particular attention to lines 6 through 10.

3.3 IntArrayTest.cs (Mod 2)

```
1    using System;
2
3    public class IntArrayTest {
4      static void Main(){
5        int[] int_array = new int[10];
6        Console.WriteLine("int_array has rank of " + int_array.Rank);
7        Console.WriteLine("int_array has " + int_array.Length + " total elements");
8        Console.WriteLine("The number of elements in the first (and only) rank is " +
9                   int_array.GetLength(0));
10       Console.WriteLine(int_array.GetType());
11
```

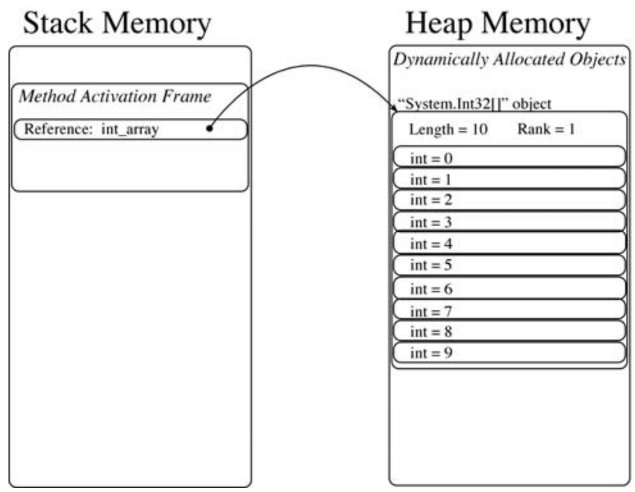



Figure 3-7: Element Values of int_array After Initialization Performed by Second `for` Loop

```
12        for(int i=0; i<int_array.GetLength(0); i++){
13           Console.Write(int_array[i] + " ");
14        }
15        Console.WriteLine();
16        for(int i=0; i<int_array.GetLength(0); i++){
17          int_array[i] = i;
18          Console.Write(int_array[i] + " ");
19        }
20         Console.WriteLine();
21      }
22    }
```

Referring to example 3.3 — lines 6 through 10 show how to use Array class methods to get information about an array. On line 6, the Rank property is accessed via the int_array reference to print out the number of int_array's dimensions. On line 7, the Length property returns the total number of array elements. On lines 8 and 9, the GetLength() method is called with an argument of 0 to determine the number of elements in the first rank. In the case of single-dimensional arrays, the Length property and GetLength(0) return the same value. On line 10, the GetType() method determines the type of the int_array reference. It returns the value "System.Int32[]," where the single pair of square brackets signifies an array type. Figure 3-8 gives the results of running this program.

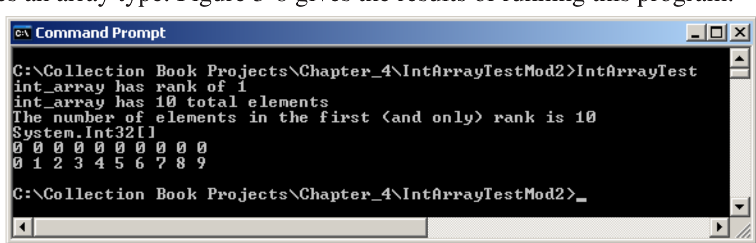


Figure 3-8: Results of Running Example 3.3

Creating Single-Dimensional Arrays Using Array Literal Values

Up to this point you have seen how memory for an array can be allocated using the `new` operator. Another way to allocate memory for an array and initialize its elements at the same time is to specify the contents of the array using *array literal* values. The length of the array is determined by the number of literal values appearing in the declaration. Example 3.4 shows two arrays being declared and created using literal values.

3.4 ArrayLiterals.cs

```
using System;

public class ArrayLiterals {
  static void Main(){
    int[] int_array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    double[] double_array = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
```



```
7
8         for(int i = 0; i < int_array.GetLength(0); i++){
9           Console.Write(int_array[i] + " ");
10        }
11        Console.WriteLine();
12        Console.WriteLine(int_array.GetType());
13        Console.WriteLine(int_array.GetType().IsArray);
14
15        Console.WriteLine();
16
17        for(int i = 0; i < double_array.GetLength(0); i++){
18          Console.Write(double_array[i] + " ");
19        }
20        Console.WriteLine();
21        Console.WriteLine(double_array.GetType());
22        Console.WriteLine(double_array.GetType().IsArray);
23      }
24    }
```

Referring to example 3.4 — the program declares and initializes two arrays using array literal values. On line 5 an array of integers named int_array is declared. The elements of the array are initialized to the values that appear between the braces. Each element's literal value is separated by a comma. The length of the array is determined by the number of literal values appearing between the braces. The length of int_array is 10.

On line 6, an array of doubles named double_array is declared and initialized with double literal values. The contents of both arrays are printed to the console. Array class methods are then used to determine the characteristics of each array and the results are printed to the console. Notice on lines 13 and 22 the use of the IsArray property. It will return true if the reference via which it is called is an array type. Figure 3-9 shows the results of running this program.

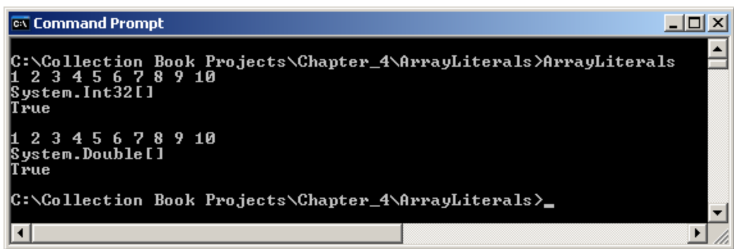


Figure 3-9: Results of Running Example 3.4

Differences Between Arrays Of Value Types And Arrays Of Reference Types

The key difference between arrays of value types and arrays of reference types is that value-type values can be directly assigned to value-type array elements. The same is not true for reference type elements. In an array of reference types, each element is a reference to an object in memory. When you create an array of references in memory you are *not* automatically creating each element's object. Instead, each reference element is automatically initialized to *null*. You must explicitly create each object you want each reference element to point to. Alternatively, the object must already exist somewhere in memory and be reachable. To illustrate these concepts, I will use an array of Objects. Example 3.5 gives the code for a short program that creates and uses an array of Objects.

3.5 ObjectArray.cs

```
using System;

public class ObjectArray {
 static void Main(){
    Object[] object_array = new Object[10];
    Console.WriteLine("object_array has type " + object_array.GetType());
    Console.WriteLine("object_array has rank " + object_array.Rank);
    Console.WriteLine();

    object_array[0] = new Object();
    Console.WriteLine(object_array[0].GetType());
    Console.WriteLine();

    object_array[1] = new Object();
    Console.WriteLine(object_array[1].GetType());
    Console.WriteLine();

    for(int i = 2; i < object_array.GetLength(0); i++){
```



```
19            object_array[i] = new Object();
20            Console.WriteLine(object_array[i].GetType());
21            Console.WriteLine();
22        }
23      }
24    }
```

Figure 3-10 shows the results of running this program.

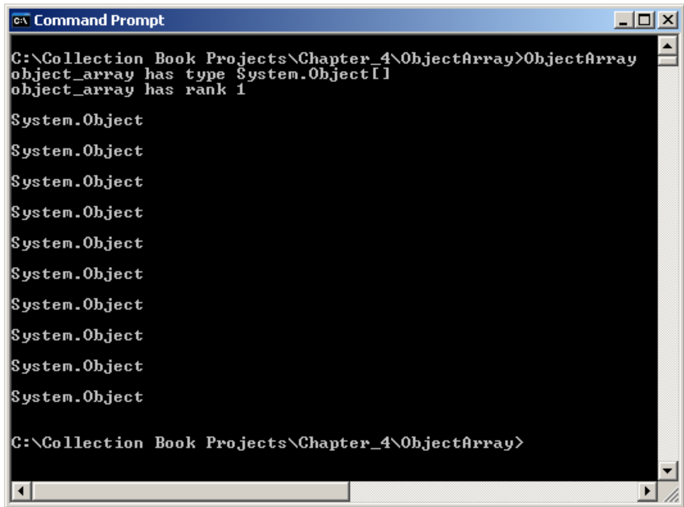


Figure 3-10: Results of Running Example 3.5

Referring to example 3.5 — on line 5, an array of Objects of length 10 is declared and created. After line 5 executes, the object_array reference points to an array of Objects in memory with each element initialized to null, as is shown in figure 3-11.

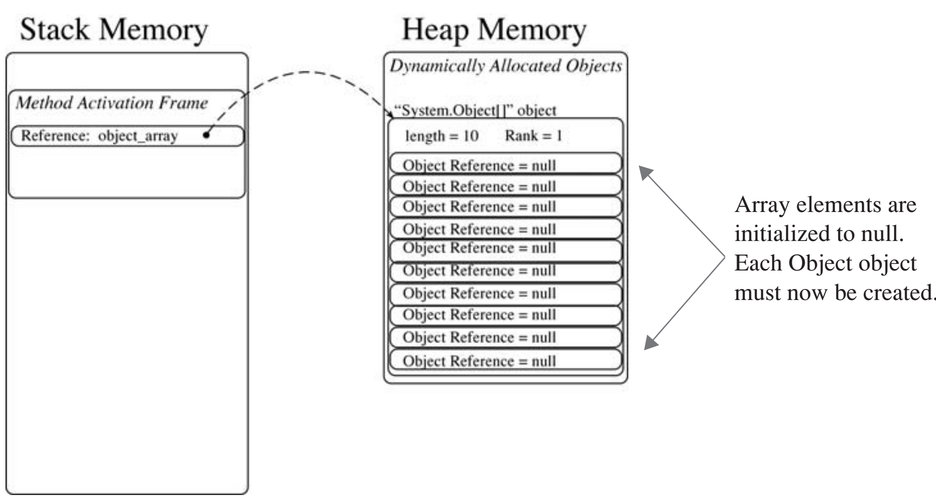


Figure 3-11: State of Affairs After Line 5 of Example 3.5 Executes

On lines 6 and 7, the program writes to the console some information about the object_array, namely, its type and rank. On line 10, a new object of type Object is created and its memory location is assigned to the Object reference located in object_array[0]. The memory picture now looks like that shown in figure 3-12. Line 11 calls the GetType() method on the object pointed to by object_array[0].

The execution of line 14 results in the creation of another object of type Object in memory. The memory picture now looks like that shown in figure 3-13. The `for` statement on line 18 creates the remaining Object objects and assigns their memory locations to the remaining object_array reference elements. Figure 3-14 shows the memory picture after the `for` statement completes execution.

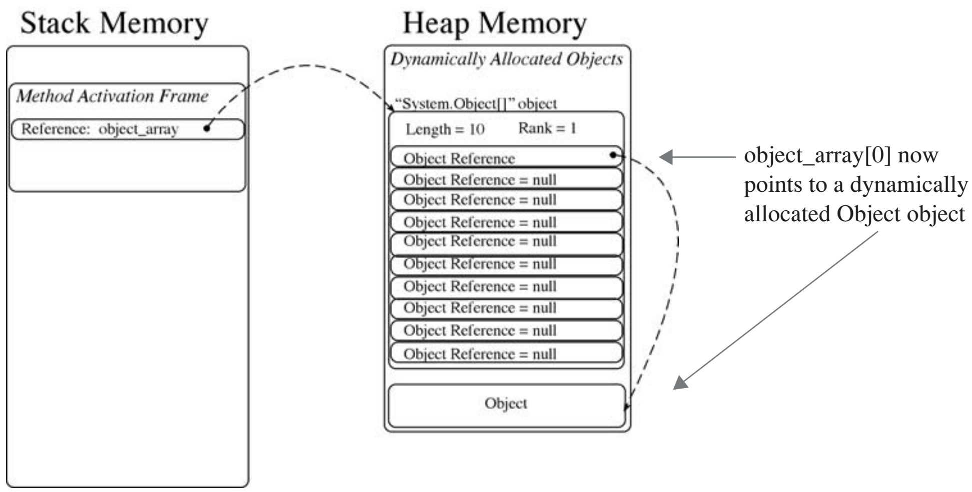


Figure 3-12: State of Affairs After Line 10 of Example 3.5 Executes.

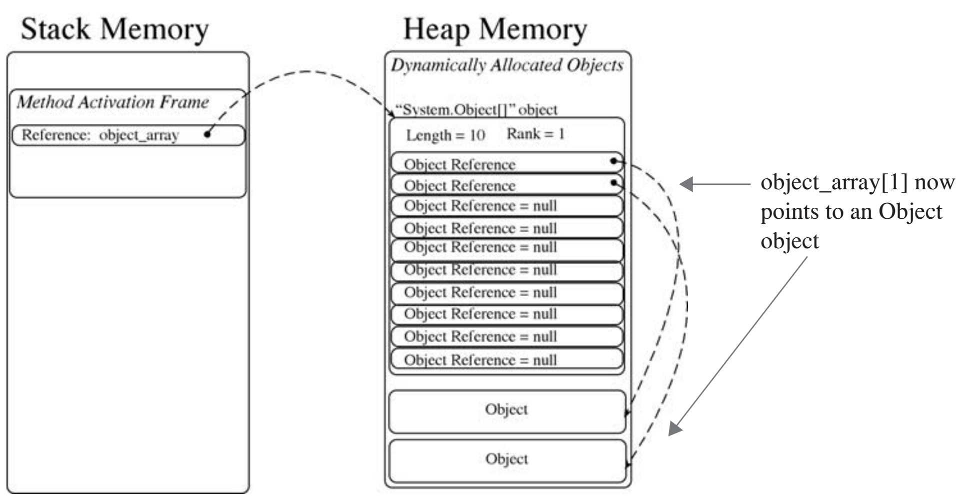


Figure 3-13: State of Affairs After Line 14 of Example 3.5 Executes

Now that you know the difference between value and reference type arrays, let's see some single-dimensional arrays being put to good use.

Single-dimensional Arrays In Action

This section offers several example programs showing how single-dimensional arrays can be used in programs. These programs represent an extremely small sampling of the usefulness arrays offer.

Message Array

One handy use for an array is to store a collection of string messages for later use in a program. Example 3.6 shows how such an array might be utilized.

3.6 MessageArray.cs

```
using System;

public class MessageArray {
  static void Main(){
    String name = null;
    int int_val = 0;

```

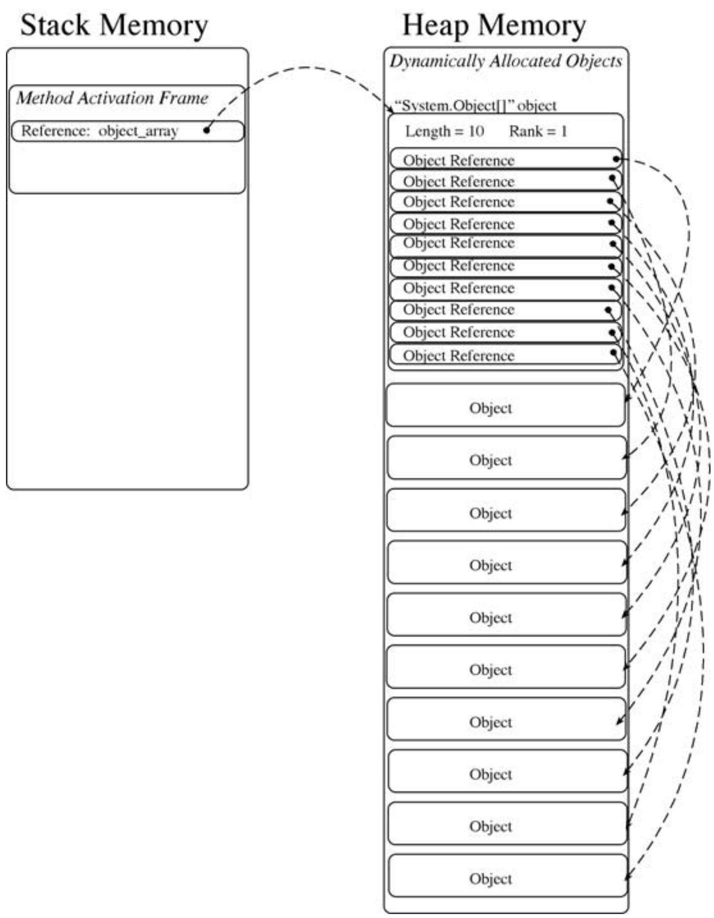



Figure 3-14: Final State of Affairs: All object_array Elements Point to an Object object

```
8         String[] messages = {"Welcome to the Message Array Program",
9                      "Please enter your name: ",
10                     ", please enter an integer: ",
11                     "You did not enter an integer!",
12                     "Thank you for running the Message Array program"};
13
14        const int WELCOME_MESSAGE     = 0;
15        const int ENTER_NAME_MESSAGE  = 1;
16        const int ENTER_INT_MESSAGE   = 2;
17        const int INT_ERROR           = 3;
18        const int THANK_YOU_MESSAGE   = 4;
19
20        Console.WriteLine(messages[WELCOME_MESSAGE]);
21        Console.Write(messages[ENTER_NAME_MESSAGE]);
22        name = Console.ReadLine();
23
24        Console.Write(name + messages[ENTER_INT_MESSAGE]);
25
26        try{
27            int_val = Int32.Parse(Console.ReadLine());
28           }catch(FormatException) { Console.WriteLine(messages[INT_ERROR]); }
29
30        Console.WriteLine(messages[THANK_YOU_MESSAGE]);
31      }
32    }
```

Referring to example 3.6 — this program creates a single-dimensional array of strings named messages. It initializes each string element using string literals. On lines 14 through 18, an assortment of constants are declared and initialized. These constants are used to index the messages array as is shown on lines 20 and 21. The program simply asks the user to enter their name followed by a request to enter an integer value. If the user fails to enter an integer, the Int32.Parse() method will throw a FormatException. Figure 3-15 shows the results of running this program.

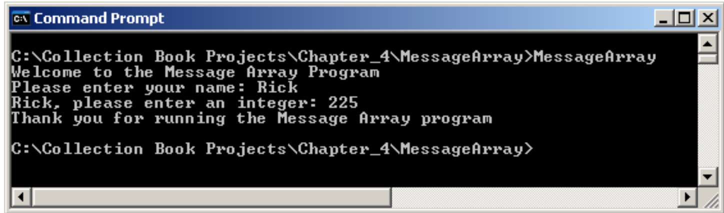


Figure 3-15: Results of Running Example 3.6

Calculating Averages

The program given in example 3.7 calculates class grade averages.

3.7 Average.cs

```
1    using System;
2
3    public class Average {
4      static void Main(){
5          double[] grades      = null;
6          double   total       = 0;
7          double   average     = 0;
8          int      grade_count = 0;
9
10         Console.WriteLine("Welcome to Grade Averager");
11         Console.Write("Please enter the number of grades to enter: ");
12         try{
13             grade_count = Int32.Parse(Console.ReadLine());
14            } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
15
16          if(grade_count > 0){
17              grades = new double[grade_count];
18                 for(int i = 0; i < grade_count; i++){
19                    Console.Write("Enter grade " + (i+1) + ": ");
20                      try{
21                         grades[i] = Double.Parse(Console.ReadLine());
22                      } catch(FormatException) { Console.WriteLine("You did not enter a number!"); }
23                 } //end for
24
25                 for(int i = 0; i < grade_count; i++){
26                    total += grades[i];
27                 } //end for
28
29                average = total/grade_count;
30                Console.WriteLine("Number of grades entered: " + grade_count);
31                Console.WriteLine("Grade average: {0:F2}           ", average);
32
33           }//end if
34      } //end main
35   }// end Average class definition
```

Referring to example 3.7 — an array reference of doubles named grades is declared on line 5 and initialized to null. On lines 6 through 8, several other program variables are declared and initialized.

The program then prompts the user to enter the number of grades. If this number is greater than 0 then it is used on line 17 to create the grades array. The program then enters a `for` loop on line 18, reads each grade from the console, converts it to a double, and assigns it to the ith element of the grades array.

After all the grades are entered into the array, the grades are summed in the `for` loop on line 25. The average is calculated on line 29. Notice how numeric formatting is used on line 31 to properly format the double value contained in the average variable. Figure 3-16 shows the results of running this program

Histogram: Letter Frequency Counter

Letter frequency counting is an important part of deciphering messages encrypted using monalphabetic substitution. Example 3.8 gives the code for a program that counts the occurrences of each letter appearing in a text string and prints the letter frequency display to the console. The program ignores all characters except the 26 letters of the alphabet.

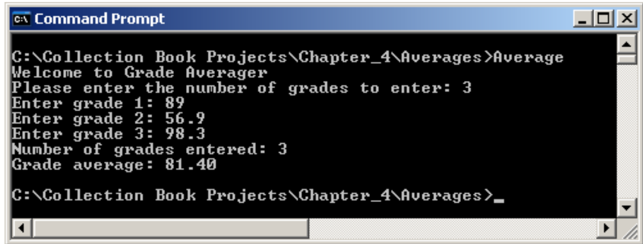


Figure 3-16: Results of Running Example 3.7

3.8 Histogram.cs

```
using System;

public class Histogram {
  static void Main(String[] args){
    int[] letter_frequencies = new int[26];
    const int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6,
              H = 7, I = 8, J = 9, K = 10, L = 11, M = 12, N = 13,
              O = 14, P = 15, Q = 16, R = 17, S = 18, T = 19, U = 20,
              V = 21, W = 22, X = 23, Y = 24, Z = 25;
    String input_string = null;

    Console.Write("Enter a line of characters: ");
    input_string = Console.ReadLine().ToUpper();


    if(input_string != null){
      for(int i = 0; i < input_string.Length; i++){
        switch(input_string[i]){
         case 'A': letter_frequencies[A]++;
                   break;
         case 'B': letter_frequencies[B]++;
                   break;
         case 'C': letter_frequencies[C]++;
                   break;
         case 'D': letter_frequencies[D]++;
                   break;
         case 'E': letter_frequencies[E]++;
                   break;
         case 'F': letter_frequencies[F]++;
                   break;
         case 'G': letter_frequencies[G]++;
                   break;
         case 'H': letter_frequencies[H]++;
                   break;
         case 'I': letter_frequencies[I]++;
                   break;
         case 'J': letter_frequencies[J]++;
                   break;
         case 'K': letter_frequencies[K]++;
                   break;
         case 'L': letter_frequencies[L]++;
                   break;
         case 'M': letter_frequencies[M]++;
                   break;
         case 'N': letter_frequencies[N]++;
                   break;
         case 'O': letter_frequencies[O]++;
                   break;
         case 'P': letter_frequencies[P]++;
                   break;
         case 'Q': letter_frequencies[Q]++;
                   break;
         case 'R': letter_frequencies[R]++;
                   break;
         case 'S': letter_frequencies[S]++;
                   break;
         case 'T': letter_frequencies[T]++;
                   break;
         case 'U': letter_frequencies[U]++;
                   break;
         case 'V': letter_frequencies[V]++;
                   break;
         case 'W': letter_frequencies[W]++;
                   break;
```



```
65           case 'X': letter_frequencies[X]++;
66                     break;
67           case 'Y': letter_frequencies[Y]++;
68                     break;
69           case 'Z': letter_frequencies[Z]++;
70                     break;
71           default : break;
72          } //end switch
73        } //end for
74
75      for(int i = 0; i < letter_frequencies.Length; i++){
76         Console.Write((char)(i + 65) + ": ");
77         for(int j = 0; j < letter_frequencies[i]; j++){
78          Console.Write('*');
79         } //end for
80         Console.WriteLine();
81      } //end for
82
83     } //end if
84   } // end main
85  } // end Histogram class definition
```

Referring to example 3.8 — on line 5, an integer array named letter_frequencies is declared and initialized to contain 26 elements, one for each letter of the English alphabet. On lines 6 through 9, several constants are declared and initialized. The constants, named A through Z, are used to index the letter_frequencies array later in the program. On line 10, a string reference named input_string is declared and initialized to null.

The program then prompts the user to enter a line of characters. The program reads this line of text and converts it to upper case using the String.ToUpper() method. Most of the work is done within the body of the `if` statement that starts on line 16. If the input_string is not null the `for` loop will repeatedly execute the `switch` statement, testing each letter of input_string and incrementing the appropriate letter_frequencies element.

Take special note on line 17 of how the length of the input_string is determined using the String class's Length property. Also note that a string's characters can be accessed using array notation. Figure 3-17 gives the results of running this program with a sample line of text.

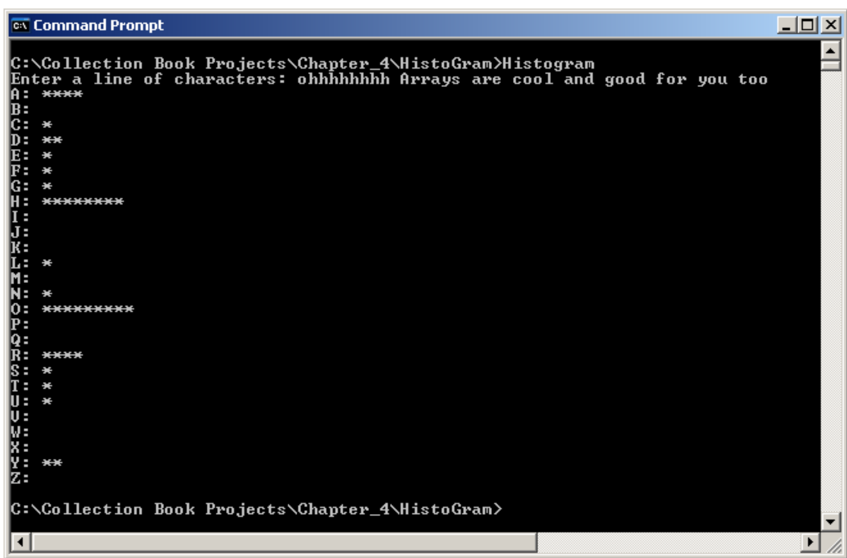


Figure 3-17: Results of Running Example 3.8

Quick Review

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument indicating the particular dimension in which you are interested. Arrays can have elements of either value or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value indicated by an integer within a set of brackets (*e.g.*, array_name[0]). Value-type element values can be directly assigned to array elements. When an array of value types is created, each element is initialized to the type's default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

Creating And Using Multidimensional Arrays

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. In this section you will learn how to create and use both kinds of multidimensional arrays. I will also show you how to create multidimensional arrays using the `new` operator as well as how to initialize multidimensional arrays using literal values.

Rectangular Arrays

A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created. Figure 3-18 gives the rectangular array declaration syntax for a two-dimensional array.

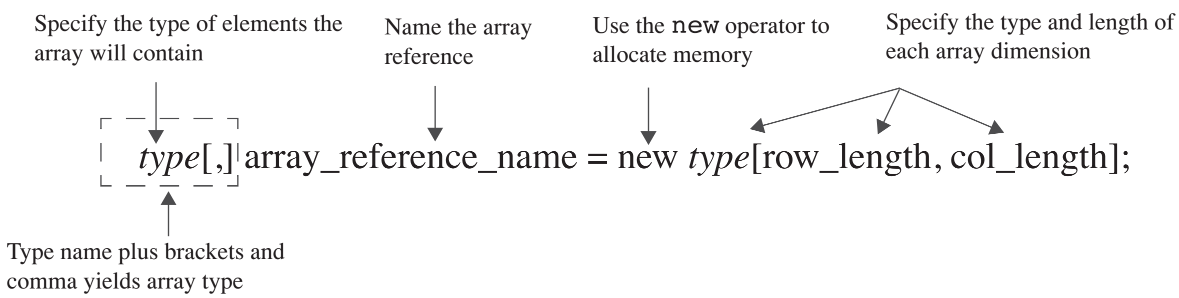


Figure 3-18: Rectangular Array Declaration Syntax

Referring to figure 3-18 — the type name combined with the brackets and comma yield the array type. For example, the following line of code declares and creates a two-dimensional rectangular array of integers having 10 rows and 10 columns:

```
int[,] int_2d_array = new int[10,10];
```

A two-dimensional array can be visualized as a grid or matrix comprised of rows and columns, as is shown in figure 3-19. Each element of the array is accessed using two index values, one each for the row and column you wish to access. For example, the following line of code would write to the console the element located in the first row, second column of int_2d_array:

```
Console.WriteLine(int_2d_array[0,1]);
```

Figure 3-19 also includes a few more examples of two-dimensional array element access. Example 3.9 offers a short program that creates a two-dimensional array of integers and prints their values to the console in the shape of a grid.

3.9 TwoDimensionalArray.cs

```
using System;

public class TwoDimensionalArray {
  static void Main(String[] args){

    try{
     int rows = Int32.Parse(args[0]);
     int cols = Int32.Parse(args[1]);

     int[,] int_2d_array = new int[rows, cols];
     Console.WriteLine("          Array rank: " + int_2d_array.Rank);
     Console.WriteLine("          Array type: " + int_2d_array.GetType());
     Console.WriteLine("Total array elements: " + int_2d_array.Length);
     Console.WriteLine();

```

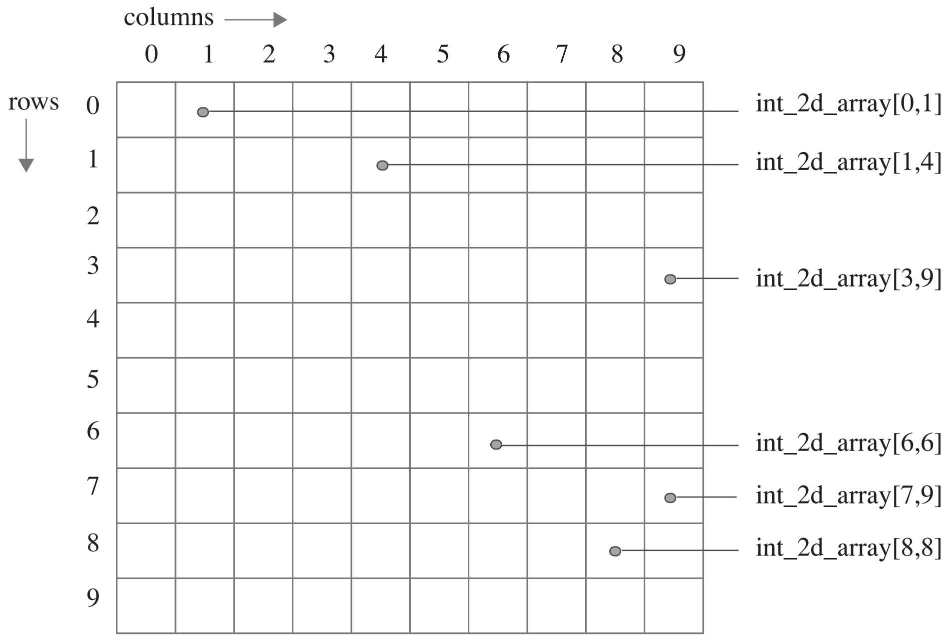



Figure 3-19: Accessing Two-Dimensional Array Elements

```
16        for(int i = 0, element = 1; i<int_2d_array.GetLength(0); i++){
17          for(int  j = 0; j<int_2d_array.GetLength(1); j++){
18           int_2d_array[i,j] = element++;
19           Console.Write("{0:D3} ",int_2d_array[i,j]);
20          }
21           Console.WriteLine();
22        }
23
24       }catch(IndexOutOfRangeException){
25          Console.WriteLine("This program requires two command-line arguments.");
26       }catch(FormatException){
27          Console.WriteLine("Arguments must be integers!");
28       }
29     }
30   }
```

Referring to example 3.9 — when the program executes, the user enters two integer values on the command line for the desired row and column lengths. These values are read and converted on lines 7 and 8, respectively. The two-dimensional array of integers is created on line 10, followed by several lines of code that writes some information about the array including its rank, type, and total number of elements to the console. The nested `for` statement beginning on line 16 *iterates* over each element of the array. Notice that the outer `for` statement on line 16 declares an extra variable named element. It's used in the body of the inner `for` loop to keep count of how many elements the array contains so that its value can be assigned to each array element. The statement on line 19 prints each array element's value to the console with the help of numeric formatting. Figure 3-20 gives the results of running this program.

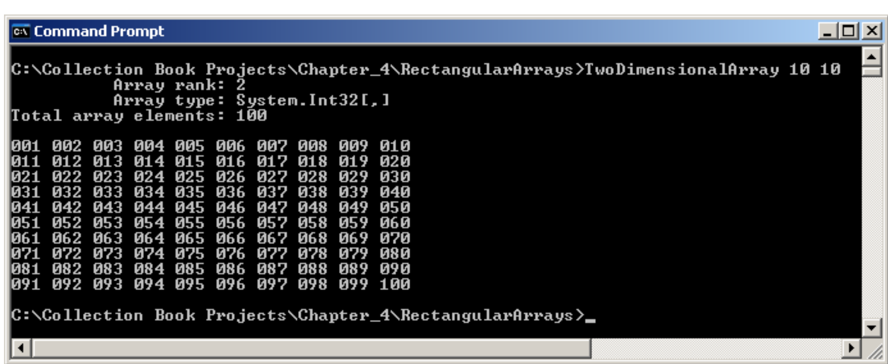


Figure 3-20: Results of Running Example 3.9

Initializing Rectangular Arrays With Array Literals

Rectangular arrays can be initialized using literal values in an array initializer expression. Study the code offered in example 3.10.

3.10 RectangularLiterals.cs

```
1    using System;
2
3    public class RectangularLiterals {
4      static void Main(){
5        char[,] char_2d_array = {{'a', 'b', 'c'},
6                                 {'d', 'e', 'f'},
7                                 {'g', 'h', 'i'}};
8
9        Console.WriteLine("char_2d_array has rank: " + char_2d_array.Rank);
10       Console.WriteLine("char_2d_array has type: " + char_2d_array.GetType());
11       Console.WriteLine("Total number of elements: " + char_2d_array.Length);
12       Console.WriteLine();
13
14       for(int i = 0; i<char_2d_array.GetLength(0); i++){
15         for(int j = 0; j<char_2d_array.GetLength(1); j++){
16           Console.Write(char_2d_array[i,j] + " ");
17         }
18         Console.WriteLine();
19       }
20     }
21   }
```

Referring to example 3.10 — a two-dimensional array of chars named char_2d_array is declared and initialized on line 5 to have 3 rows and 3 columns. Notice how each row of characters appears in a comma-separated list between a set of braces. Each row of initialization data is itself separated from the next row by a comma, except for the last row of data on line 7. Lines 9 through 11 write some information about the character array to the console, namely, its rank, type, and total number of elements. The nested `for` statement beginning on line 14 iterates over the array and prints each character to the console in the form of a grid. Figure 3-21 shows the results of running this program.

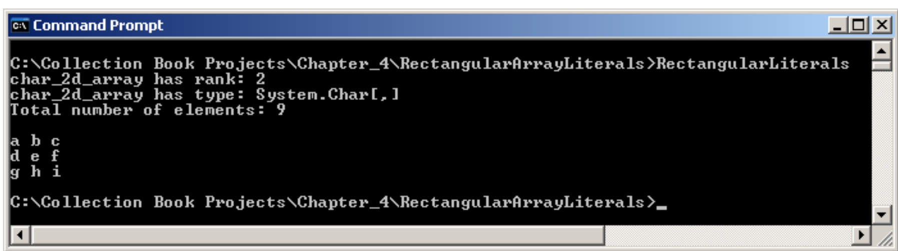


Figure 3-21: Results of Running Example 3.10

Ragged Arrays

A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be dynamically created during program execution, resulting in the possibility that the array dimensions may differ in length, hence the name ragged array. The Length property returns the number of array elements declared in the leftmost dimension.

Figure 3-22 shows the ragged array declaration syntax for a two-dimensional ragged array. Example 3.11 gives a short program showing the use of a ragged array.

3.11 Ragged2dArray.cs

```
using System;

public class Ragged2dArray {
  static void Main(){
    int[][] ragged_2d_array = new int[10][];

    Console.WriteLine("ragged_2d_array has rank: " + ragged_2d_array.Rank);
    Console.WriteLine("ragged_2d_array has type: " + ragged_2d_array.GetType());
    Console.WriteLine("Total number of elements: " + ragged_2d_array.Length);
    Console.WriteLine();

    for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
```



```
13        ragged_2d_array[i] = new int[i+1];
14      }
15
16      for(int i = 0; i<ragged_2d_array.GetLength(0); i++){
17        for(int j = 0; j<ragged_2d_array[i].GetLength(0); j++){
18          Console.Write(ragged_2d_array[i][j] + " ");
19        }
20        Console.WriteLine();
21      }
22    }
23  }
```

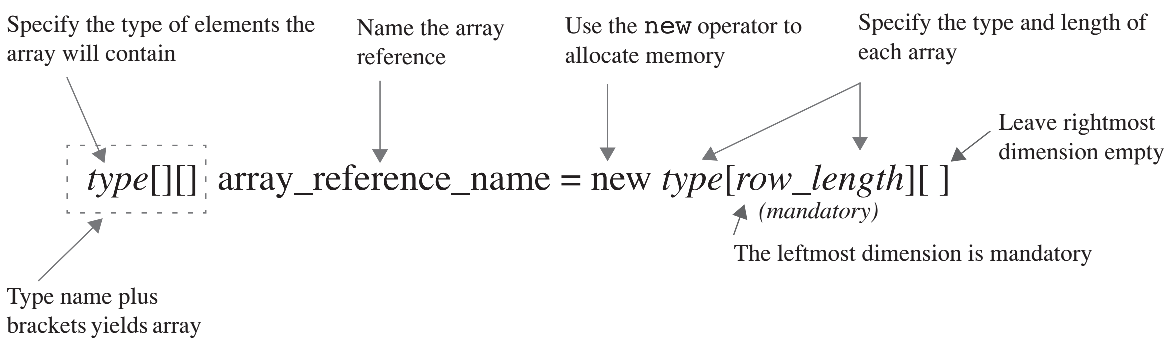


Figure 3-22: Array Declaration Syntax for a Two-Dimensional Ragged Array

Referring to example 3.11 — on line 5 a two-dimensional ragged array of integers is declared and created. Lines 7 through 9 write some information about the array including its rank, type, and total number of elements to the console. The `for` statement beginning on line 12 creates 10 new arrays of varying lengths and assigns their references to each element of ragged_2d_array. The next `for` statement on line 16 iterates over the ragged two-dimensional array structure and writes the value of each element to the console. Figure 3-23 shows the results of running this program.

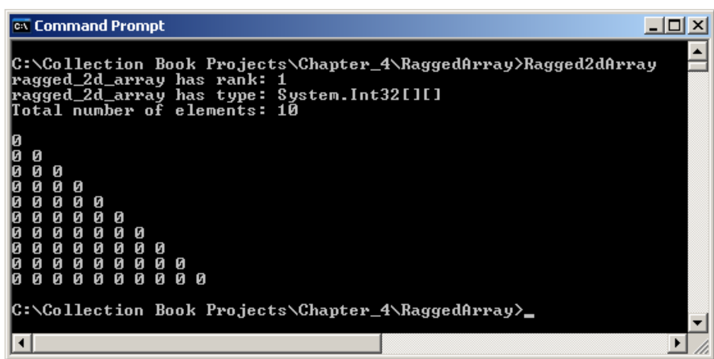


Figure 3-23: Results of Running Example 3.11

Multidimensional Arrays In Action

The example presented in this section shows how single and multidimensional arrays can be used together effectively.

Weighted Grade Tool

Example 3.12 gives the code for a class named WeightedGradeTool. The program calculates a student's final grade based on weighted grades.

3.12 WeightedGradeTool.cs

```
using System;

public class WeightedGradeTool {
    static void Main() {

```



```
        double[,] grades = null;
        double[] weights = null;
        String[] students = null;
        int student_count = 0;
        int grade_count = 0;
        double final_grade = 0;

        Console.WriteLine("Welcome to Weighted Grade Tool");

        /**************** get student count *********************/
        Console.Write("Please enter the number of students: ");
        try {
            student_count = Int32.Parse(Console.ReadLine());
        }
        catch (FormatException) {
            Console.WriteLine("That was not an integer!");
            Console.WriteLine("Student count will be set to 3.");
            student_count = 3;
        }


        if (student_count > 0) {
            students = new String[student_count];
            /***************** get student names **********************/
            for (int i = 0; i < students.Length; i++) {
                Console.Write("Enter student name: ");
                students[i] = Console.ReadLine();
            }

            /**************** get number of grades per student **********/
            Console.Write("Please enter the number of grades to average: ");
            try {
                grade_count = Int32.Parse(Console.ReadLine());
            }
            catch (FormatException) {
                Console.WriteLine("That was not an integer!");
                Console.WriteLine("Grade count will be set to 3.");
                grade_count = 3;
            }

            /****************** get raw grades *****************************/
            grades = new double[student_count, grade_count];
            for (int i = 0; i < grades.GetLength(0); i++) {
                Console.WriteLine("Enter raw grades for " + students[i]);
                for (int j = 0; j < grades.GetLength(1); j++) {
                    Console.Write("Grade " + (j + 1) + ": ");
                    try {
                        grades[i, j] = Double.Parse(Console.ReadLine());
                    }
                    catch (FormatException) {
                        Console.WriteLine("That was not a double!");
                        Console.WriteLine("Grade will be set to 100");
                        grades[i, j] = 100;
                    }
                }//end inner for
            }

            /***************** get grade weights *********************/
            weights = new double[grade_count];
            Console.WriteLine("Enter grade weights. Make sure they total 100%");
            for (int i = 0; i < weights.Length; i++) {
                Console.Write("Weight for grade " + (i + 1) + ": ");
                try {
                    weights[i] = Double.Parse(Console.ReadLine());
                }
                catch (FormatException) {
                    Console.WriteLine("That was not a double!");
                    Console.WriteLine("The weight will be set to .25");
                    weights[i] = .25;
                }
            }

            /****************** calculate weighted grades ********************/
            for (int i = 0; i < grades.GetLength(0); i++) {
                for (int j = 0; j < grades.GetLength(1); j++) {
                    grades[i, j] *= weights[j];
                }//end inner for
            }

            /***************** calculate and print final grade ********************/
            for (int i = 0; i < grades.GetLength(0); i++) {
```



```
                Console.WriteLine("Weighted grades for " + students[i] + ": ");
                final_grade = 0;
                for (int j = 0; j < grades.GetLength(1); j++) {
                    final_grade += grades[i, j];
                    Console.Write(grades[i, j] + " ");
                }//end inner for
                Console.WriteLine(students[i] + "'s final grade is: " + final_grade);
            }
        }// end if
    }// end Main
}// end class
```

Figure 3-24 shows the results of running this program.

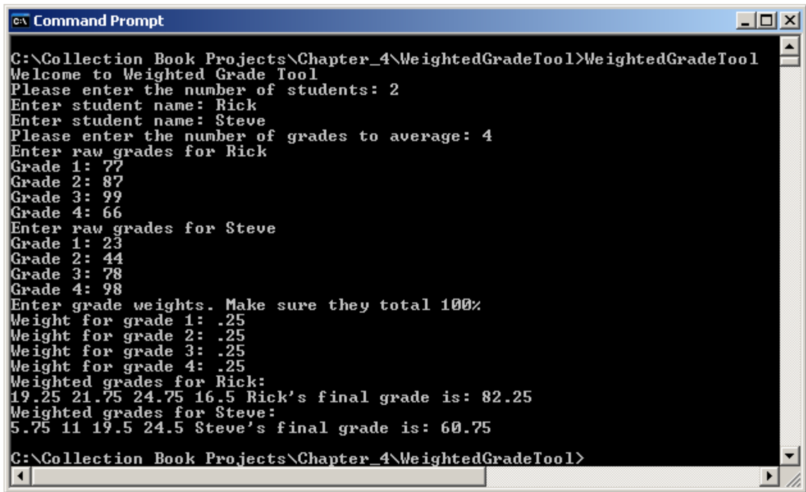


Figure 3-24: Results of Running Example 3.12

Quick Review

C# supports two kinds of multidimensional arrays: *rectangular* and *ragged*. A *rectangular array* is a multidimensional array whose shape is fixed based on the length of each *dimension* or *rank*. All of a rectangular array's dimensions must be specified when the array object is created. A *ragged array* is an array of arrays. Ragged arrays can be any number of dimensions, but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.

The Main() Method's String Array

Now that you have a better understanding of arrays, the Main() method's string array should make more sense. This section explains the purpose and use of the Main() method's string array.

Purpose And Use Of The Main() Method's String Array

The purpose of the Main() method's string array is to enable C# applications to accept and act upon command-line arguments. The csc compiler is an example of a program that takes command-line arguments, the most important of which is the name of the file to compile. This chapter gave several examples of accepting program input via the command line. Now that you are armed with a better understanding of how arrays work, you have the knowledge to write programs that accept and process command-line arguments.

Example 3.13 gives a short program that accepts a line of text as a command-line argument and displays it in lower or upper case depending on the first command-line argument.

3.13 CommandLine.cs

```
using System;
using System.Text;

public class CommandLine {
  static void Main(String[] args){
    StringBuilder sb = null;
    bool upper_case = false;
    int start_index = 0;

   /********** check for upper case option **************/
   if(args.Length > 0){
      switch(args[0][0]){ // get the first character of the first argument
        case '-' :
                 if(args[0].Length > 1){
                   switch(args[0][1]){ // get the second character of the first argument
                      case 'U' :
                      case 'u' : upper_case = true;
                                 break;
                      default:   upper_case = false;
                                 break;
                   }
                  }
                 start_index = 1;
                 break;
        default: upper_case = false;
                 break;

      }// end outer switch

      sb = new StringBuilder();   //create StringBuffer object

   /******* process text string **********************/
     for(int i = start_index; i < args.Length; i++){
           sb.Append(args[i] + " ");
     }//end for

      if(upper_case){

        Console.WriteLine(sb.ToString().ToUpper());
      }else {

        Console.WriteLine(sb.ToString().ToLower());
       }//end if/else

   } else { Console.WriteLine("Usage: CommandLine [-U | -u] Text string");}

  }//end main
}//end class
```

Figure 3-25 shows the results of running this program.

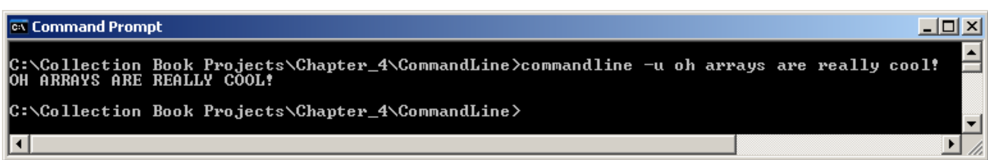


Figure 3-25: Results of Running Example 3.13

Manipulating Arrays With The System.Array Class

The .NET platform makes it easy to perform common array manipulations such as searching and sorting with the System.Array class. Example 3.14 offers a short program that shows the Array class in action sorting an array of integers.

3.14 ArraySortApp.cs

```
   using System;

public class ArraySortApp {
  static void Main() {
    int[] int_array = { 100, 45, 9, 1, 34, 22, 6, 4, 3, 2, 99, 66 };

```



```
      for (int i = 0; i < int_array.Length; i++) {
          Console.Write(int_array[i] + " ");
      }
      Console.WriteLine();

      Array.Sort(int_array);

      for (int i = 0; i < int_array.Length; i++) {
          Console.Write(int_array[i] + " ");
      }
      } // end Main() method
  } // end ArraySortApp class definition
```

Figure 3-26 shows the results of running this program.

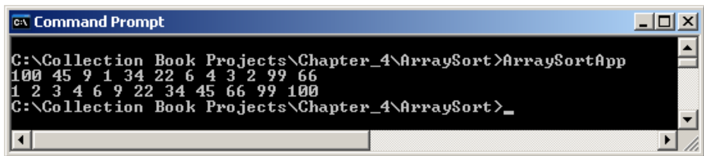


Figure 3-26: Results of Running Example 3.14

Non Supported IList Operations

Although the System.Array class implements the IList and IList<T> interfaces, an array is a fixed size and does not grow automatically to accept new elements. Because of this fixed-size characteristic there are four members of the IList and IList<T> interfaces that are not supported and any attempt to use them will throw a NonSupportedException. These include the Add(), Insert(), Remove(), and RemoveAt() methods.

Summary

C# array types have special functionality because of their special inheritance hierarchy. C# array types directly inherit functionality from the System.Array class and implement the ICloneable, IList, ICollection, and IEnumerable interfaces. Arrays are also serializable.

Single-dimensional arrays have one dimension — length. You can get an array's length by calling the GetLength() method with an integer argument that indicates the dimension in which you are interested. You can also get the length of a single dimensional array by accessing its *Length* property. Arrays can have elements of either value or reference types. An array type is created by specifying the type name of array elements followed by one set of brackets [ ]. Use System.Array class methods and properties to get information about an array.

Each element of an array is accessed via an index value contained within a set of brackets. Value-type element values can be directly assigned to array elements. When an array of value types is created, each element is initialized to the types default value. Each element of an array of references is initialized to null. Each object that a reference element points to must either already exist or be created during program execution.

C# supports two kinds of multidimensional arrays: rectangular and ragged. A rectangular array is a multidimensional array whose shape is fixed based on the length of each dimension or rank. All of a rectangular array's dimensions must be specified when the array object is created.

A ragged array is an array of arrays. Ragged arrays can be any number of dimensions but the last, or rightmost, dimension is omitted from the array creation expression. Each rightmost array object must then be created during program execution, introducing the possibility that the array's dimensions may differ in length.

Use the built-in methods and properties of the System.Array class to perform certain array manipulations such as sorting.

References

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Rick Miller. C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming. ISBN-13: 978-1-932504-07-1. Pulp Free Press

Donald E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching,* Second Edition. Addison-Wesley. Reading Massachusetts. ISBN: 0-201-89685-0

Notes

Chapter 4

Contax T

Musee de L'Armee

Fundamental Data Structures

Learning Objectives

- *Describe the performance characteristics of fundamental data structures used to implement collections*
- *Learn how to choose a collection based on its underlying data structure*
- *Describe the performance characteristics of an array*
- *Describe the performance characteristics of a linked list*
- *Describe the performance characteristics of a hash table*
- *Describe the performance characteristics of a red-black tree*
- *Describe the performance characteristics of a stack*
- *Describe the performance characteristics of a queue*

Introduction

In this chapter, I want to introduce you to the performance characteristics of several different types of foundational data structures. These include the *array*, *linked list*, *hash table*, and *red-black binary tree*. Knowing a little bit about how these data structures work and behave will make it easier for you to select the .NET collection type that's best suited for your particular application.

Array Performance Characteristics

As you know already from reading Chapter 3, an array is a contiguous collection of homogeneous elements. You can have arrays of value types or arrays of references to objects. The general performance issues to be aware of regarding arrays concern inserting new elements into the array at some position prior to the last element, accessing elements, and searching for particular values within the array.

When a new element is inserted into an array at a position other than the end, room must be made at that index location for the insertion to take place by shifting the remaining references one element to the right. This series of events is depicted in figures 4-1 through 4-3.

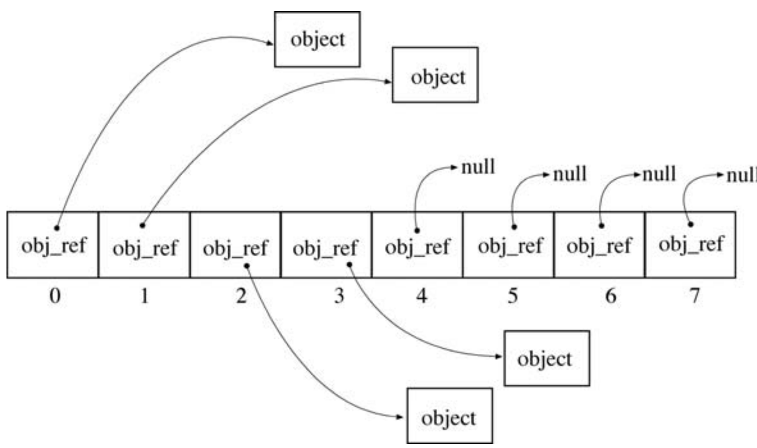


Figure 4-1: Array of Object References Before Insertion

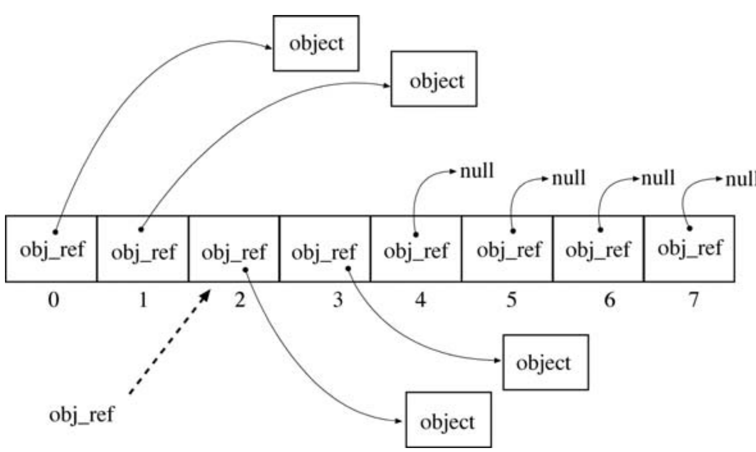


Figure 4-2: New Reference to be Inserted at Array Element 3 (index 2)

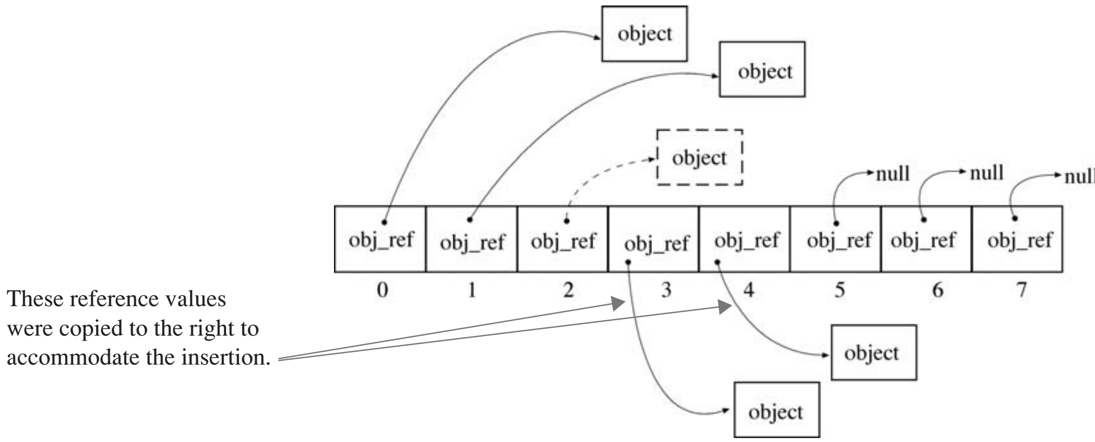


Figure 4-3: Array After New Reference Insertion

Referring to figures 4-1 through 4-3 — an array of object references contains references that may point to an object or to null. In this example, array elements 1 through 4 (index values 0 through 3) point to objects while the remaining array elements point to null.

A reference insertion is really just an assignment of the value of the reference being inserted to the reference residing at the target array element. To accommodate the insertion, the values contained in references located to the right of the target element must be reassigned one element to the right. (*i.e.,* They must be shifted to the right.) It is this shifting action that causes a performance hit when inserting elements into an array-based collection. If the insertion triggers the array growth mechanism, then you'll receive a double performance hit. The insertion performance penalty, measured in time, grows with the length of the array. Element retrieval, on the other hand, takes place fairly quickly because of the way array element addresses are computed. (*Refer to Chapter 3 — Arrays*)

Linked List Performance Characteristics

A linked list is a data structure whose elements stand alone in memory. (And may indeed be located anywhere in the heap!) Each element is linked to another by a reference. Unlike the elements of an array, which are ordinary references, each linked list node is a complex data structure that contains a reference to the *previous* node in the list, the *next* node in the list, and a reference to an object payload, as figure 4-4 illustrates.

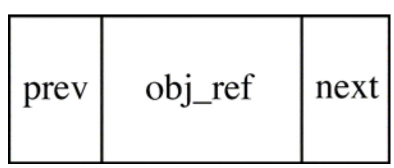


Figure 4-4: Linked List Node Organization

Whereas an array's elements are always located one right after the other in memory, and their memory addresses quickly calculated, a linked list's elements can be, and usually are, scattered in memory hither and yonder. The nice thing about linked lists is that element insertions take place fairly quickly because no element shifting is required. Figures 4-5 through 4-8 show the sequence of events for the insertion of a circular linked list node. Referring to figures 4-5 through 4-8 — a linked list contains one or more non-contiguous nodes. A node insertion requires reference rewiring. This entails setting the *previous* and *next* references on the new node in addition to resetting the affected references of its adjacent list nodes.

If this is your first exposure to this type of programming it can look complicated. As with practically everything in the programming world, if you stare at it long enough it will start to make sense.

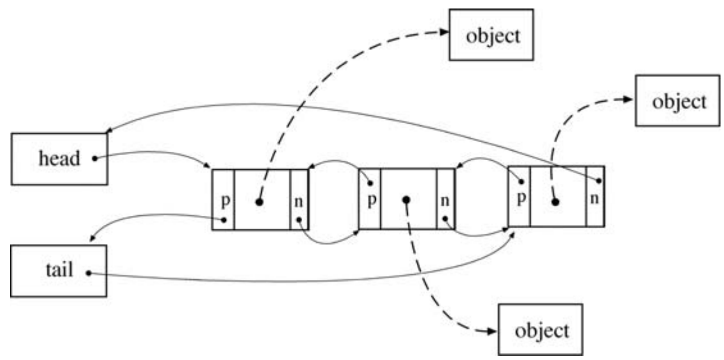


Figure 4-5: Linked List Before New Element Insertion

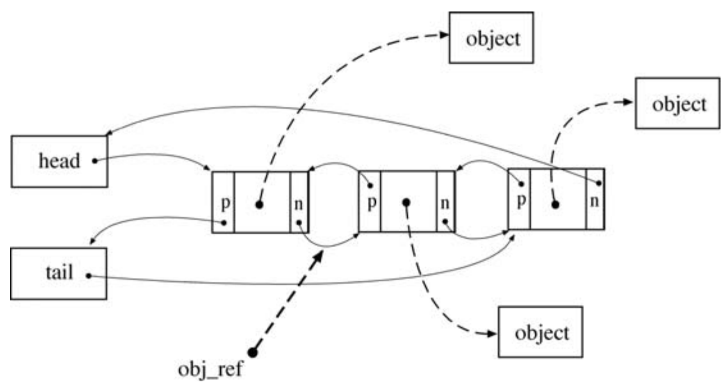


Figure 4-6: New Reference Being Inserted Into Second Element Position

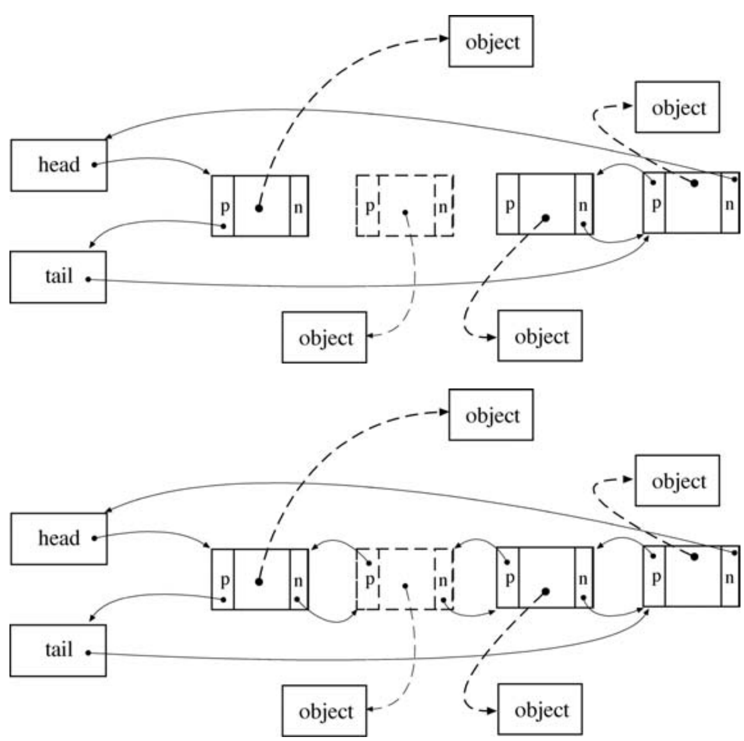


Figure 4-7: References of Previous, New, and Next List Elements must be Manipulated

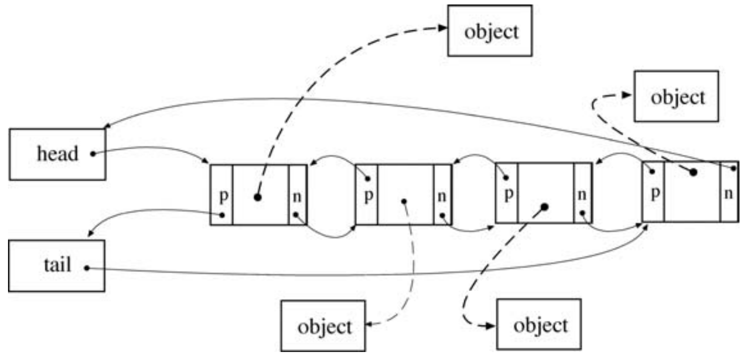


Figure 4-8: Linked List Insertion Complete

Hash Table Performance Characteristics

A hash table is an array whose elements can point to a series of nodes. Structurally, as you'll see, a hash table is a cross between an array and a one-way linked list. In an ordinary array, elements are inserted by index value. If there are potentially many elements to insert, the array space required to hold all the elements would be correspondingly large as well. This may result in wasted memory space. The hash table addresses this problem by reducing the size of the array used to point to its elements and assigning each element to an array location based on a *hash function* as figure 4-9 illustrates.

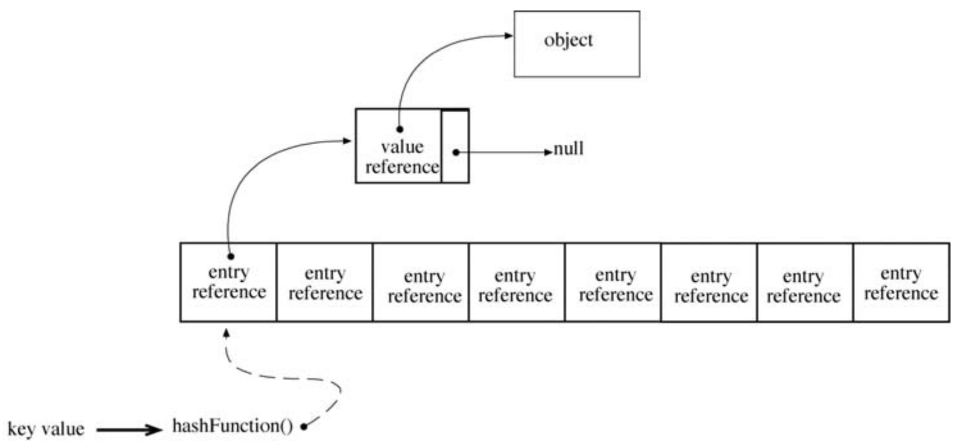


Figure 4-9: A Hash Function Transforms a Key Value into an Array Index

Referring to figure 4-9 — the purpose of the hash function is to transform the key value into a unique array index value. However, sometimes two unique key values translate to the same index value. When this happens a *collision* is said to have occurred. The problem is resolved by chaining together nodes that share the same hash table index as is shown in figure 4-10.

The benefits of a hash table include lower initial memory overhead and relatively fast element insertions. On the other hand, if too many insertion collisions occur, the linked elements must be traversed to insert new elements or to retrieve existing elements. List traversal extracts a performance penalty.

Chained Hash Table vs. Open-Address Hash Table

The hash table discussed above is referred to as a *chained hash table*. Another type of hash table, referred to as an *open-address hash table*, uses a somewhat larger array and replaces the linking mechanism with a *slot probe function* that searches for empty space when the table approaches capacity.

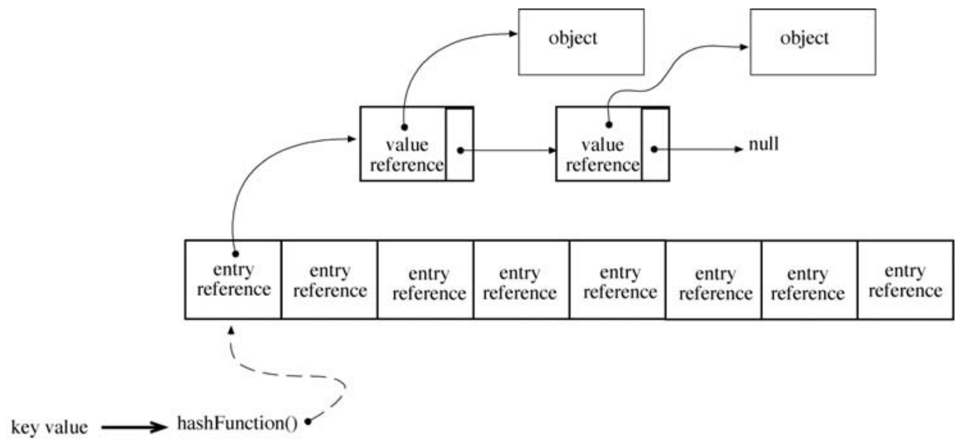


Figure 4-10: Hash Table Collisions are Resolved by Linking Nodes Together

Red-Black Tree Performance Characteristics

A *red-black tree* is a special type of *binary search tree* with a self-balancing characteristic. *Tree nodes* have an additional data element, *color*, that is set to either red or black. The data elements of a red-black tree node are shown in figure 4-11.

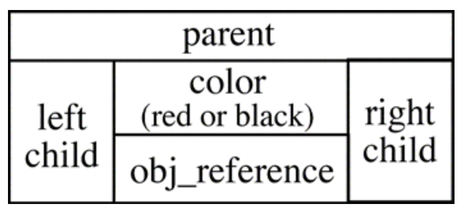


Figure 4-11: Red-Black Tree Node Data Elements

Insertions into a red-black tree are followed by a self-balancing operation. This ensures that all leaf nodes are the same number of black nodes away from the root node. Figure 4-12 shows the state of a red-black tree after inserting the integer values 1 through 9 in the following insertion order: 9, 3, 5, 6, 7, 2, 8, 4, 1. (Red nodes are shown lightly shaded.)

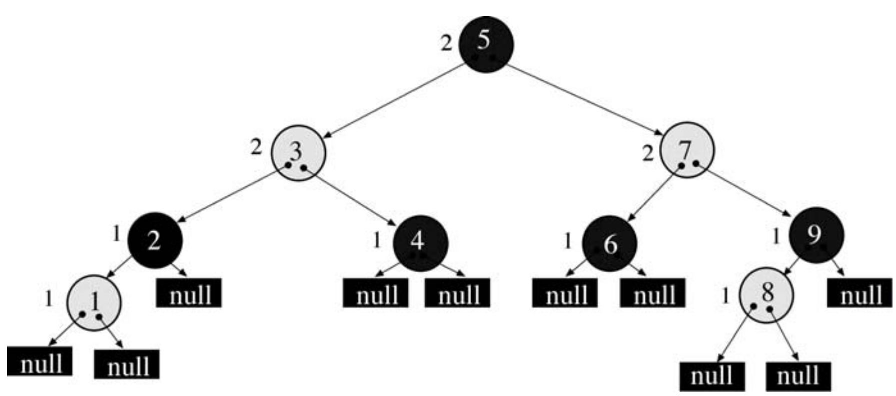


Figure 4-12: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1

Referring to figure 4-12 — the numbers appearing to the left of each node represent the height of the tree in black nodes. The primary benefit associated with a red-black tree is the generally overall good node search performance

regardless of the number of nodes the tree contains. However, because the tree reorders itself with each insertion, an insertion into a tree that contains lots of nodes incurs a performance penalty.

Think of it in terms of a clean room versus a messy room. You can store things really fast in a messy room because you just throw your stuff anywhere. Finding things in a messy room takes some time. You may have to look high and low before finding what you're looking for. Storing things in a clean room, conversely, takes a little while longer, but when you need something, you can find if fast!

Stacks And Queues

Two additional data structures you'll encounter in the collections API are *stacks* and *queues*. A stack is a data structure that stores objects in a *last-in-first-out* (LIFO) basis. Objects are placed on the stack with a *push* operation and removed from the stack with a *pop* operation. A stack operates like a plate dispenser, where you put in a stack of plates and take plates off the stack one at a time. The last plate inserted into the plate dispenser is the first plate dispensed when someone needs a plate. Figure 4-13 shows the state of a stack after several pushes and pops.

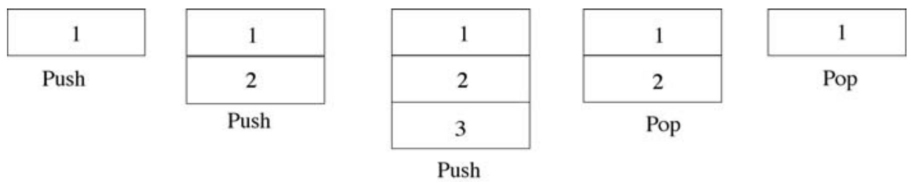


Figure 4-13: A Stack After Several Push and Pop Operations

A *queue* is a data structure that stores objects in a *first-in-first-out* (FIFO) basis. A queue operates like a line of people waiting for some type of service; the first person in line is the first person to be served. People arriving in line must wait for service until they reach the front of the line. Objects are added to a queue with an *enqueue* operation and removed with a *dequeue* operation. Figure 4-14 shows the state of a queue after several enqueues and dequeues.

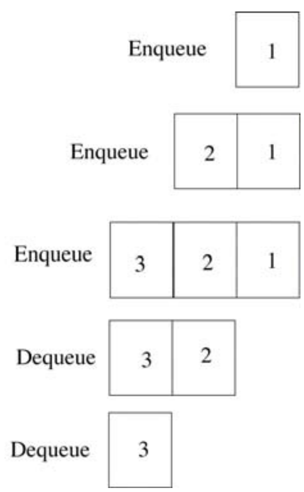


Figure 4-14: A Queue After Several Enqueue and Dequeue Operations

Summary

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A *hash function* performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A red-black tree is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access times for a red-black tree-based collection is fairly quick.

References

Microsoft Developer Network (MSDN) .NET Framework 3.5 Documentation [http://www.msdn.com]

John Franco. *Red-Black Tree Demonstration Applet*. [http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html]

Thomas H. Cormen, et. al. *Introduction To Algorithms*. The MIT Press, Cambridge, MA. ISBN: 0-262-03141-8

Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, *Third Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*, *Second Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Rick Miller. C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming. ISBN-13: 978-1-932504-07-1. Pulp Free Press.

Notes

Chapter 5

Yashica Mat 124G

Ghost Walker

Understanding Generics

Learning Objectives

- State the benefits of using generic collection types vs. non-generic collection types
- Use generic type parameters to create generic types and methods
- State the purpose of generic type parameters and generic type arguments
- State the limitations of unbounded type parameters
- List four different types that can be generic types
- List and describe six generic type constraints
- List and describe the interfaces targeted by generic collection classes

Introduction

This chapter offers you a peek under the covers to reveal the inner workings of generic types and methods. Programmers unfamiliar with the .NET framework and with generic types specifically are initially bewildered by the confusing syntax used in the generic collection classes. What exactly does the ‘T’ represent in a List<T> collection class? Another question I hear frequently is “How does the List<T> class know how to manipulate different types when performing operations like sorting?” When you finish reading this chapter you’ll know the answer to these questions and many more.

I’ll start by showing you how to create generic types and methods using single and multiple generic, unbounded type parameters. Next, you’ll learn how to apply type constraints when defining generic types. Once you have a good understanding of generic types, I’ll explain the benefits of using generic types vs. non-generic types.

Fundamentally, this chapter prepares you for a formal encounter with generic collection types later in the book. Let’s get to work!

Creating Generic Types

A *generic type* (class, structure, interface, or delegate) is one that’s declared with the help of one or more *type parameters*. A type parameter serves as a place holder which is ultimately replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a type *template*; the purpose of the template is to create new types as if stamping them from a mold. You can create generic types using single or multiple type parameters. In this section I will focus the discussion on the creation of generic classes. I will postpone the discussion of other generic types (interfaces, structs, and delegates) until later in the book.

Let’s start by creating a generic class that contains a single type parameter in its definition.

Using a Single Type Parameter

Example 5.1 gives the code for a generic class that uses a single type parameter in its definition.

5.1 SimpleGeneric.cs

```
1      using System;
2
3      public class SimpleGeneric<T> {
4
5        public void PrintValue(T arg){
6          Console.WriteLine(arg);
7        }
8      }
```

Referring to example 5.1 — the SimpleGeneric class has one *type parameter* signified by the character T that appears within the angle brackets “< >”. You can use any identifier name to signify the type parameter, but in the .NET generic framework you’ll generally see single-character uppercase letters used for this purpose.

When an instance of SimpleGeneric is created, the *type argument* supplied in place of T is substituted for T wherever it appears in the code. In this example the character T appears in the parameter list of the PrintValue() method. Example 5.2 shows the SimpleGeneric class in action.

5.2 MainApp.cs

```
1      public class MainApp {
2        public static void Main(){
3          SimpleGeneric<string> sg_1 = new SimpleGeneric<string>();
4          sg_1.PrintValue("Hello World");
5        }
6      }
```

Referring to example 5.2 — an instance of SimpleGeneric is created on line 3 by supplying the string type as a type argument. This has the effect of creating a new type (e.g., SimpleGeneric<string>). The string type replaces T in the definition of the PrintValue() method (See example 5.1, line 5). Figure 5-1 shows the results of running this program.

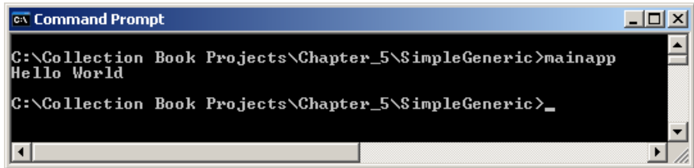


Figure 5-1: Results of Running Example 5.2

Using Multiple Type Parameters

You can create generic types that use multiple type parameters. These type parameters can function as placeholders in methods, fields, properties, and other class member definitions. Example 5.3 gives the code for a class that uses two type parameters in its definition.

5.3 TwoParameterGeneric.cs

```
using System;

public class TwoParameterGeneric<T, U> {

  //fields
  private T _field1;
  private U _field2;

  //constructors
  public TwoParameterGeneric(T arg1, U arg2){
    _field1 = arg1;
    _field2 = arg2;
  }

  private TwoParameterGeneric(){
    // effectively disable the default constructor
  }

  //properties
  public T PropertyOne {
    get { return _field1; }
    set { _field1 = value; }
  }

  public U PropertyTwo {
    get { return _field2; }
    set { _field2 = value; }
  }

  public U PrintValue(){
    Console.WriteLine("T is a " + (_field1.GetType()).ToString() + " with value: " + _field1);
    Console.WriteLine("U is a " + (_field2.GetType()).ToString() + " with value: " + _field2);
    return _field2;
  }
}
```

Referring to example 5.3 — the TwoParameterGeneric class declares two type parameters named T and U respectively. These type parameters appear throughout the code in field, constructor, property, and method definitions. Example 5.4 shows this class in action.

5.4 MainApp.cs

```
1       using System;
2
3       public class MainApp {
4         public static void Main(){
5           TwoParameterGeneric<string, int> tpg_1 = new TwoParameterGeneric<string, int>("Hello World", 4);
6           Console.WriteLine(tpg_1.PrintValue());
7           tpg_1.PropertyOne = "Second string";
8           tpg_1.PropertyTwo = 378;
9           Console.WriteLine("-----------------------");
10          Console.WriteLine(tpg_1.PrintValue());
11        }
12      }
```

Referring to example 5.4 — an instance of the TwoParameterGeneric class is created on line 5 using the types string and int as type arguments. The string "Hello World" and the integer value 4 are passed as arguments to the constructor. Line 6 makes a call to the PrintValue() method and writes its return value to the console. Lines 7 and 8 dem-

onstrate the use of the properties and again on line 10 the PrintValue() method is called and its return value printed to the console. Figure 5-2 shows the results of running this program.

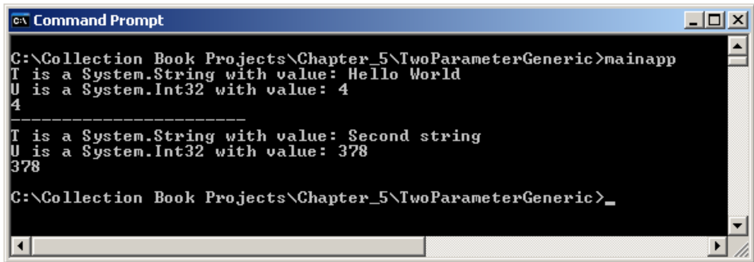


Figure 5-2: Results of Running Example 5.4

Unbounded Type Parameters

The previous examples used *unbounded type parameters* in the definition of generic classes. There's not much you can do in the code with an unbounded type parameter. The reason why is that when presented with an unspecified interface for a type parameter, the compiler can only assume you mean to target the System.Object interface, which results in a very limited range of operations. That's why the examples I provide in this section do nothing more than print string values to the console via an object's ToString() method. Since every type (value, reference, delegate, etc.) ultimately extends from System.Object, I can safely write code in the body of my generic type examples that targets the System.Object interface. I can transcend this limitation by specifying a particular targeted interface via a *constraint*. You'll learn how to apply constraints to generic types later in this chapter, but first I want to show you how to create generic methods.

Quick Review

A *generic type* is one that's declared with the help of one or more *type parameters*. A type parameter serves as a place holder which will eventually be replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a type *template*; the purpose of the template is to create new types as if stamping them from a mold.

You can create generic types that use one or more type parameters. Type parameters can appear in the definition of any type member, including fields, constructors, properties, methods, etc.

In the absence of a type parameter *constraint*, the compiler assumes the targeted interface will be that of the System.Object class. An unconstrained type parameter is referred to as an *unbounded type parameter.*

Creating Generic Methods

A *generic method* is defined with the help of one or more type parameters that appear inside of angle brackets "< >". A generic method can appear in the definition of an ordinary, non-generic class or structure. That is, it's not a requirement for a class or structure to be generic for it to contain generic method definitions. Also, generic methods can define single or multiple type parameters. I say we fling ourselves into the deep end of the swimming pool and take a look at a generic method that uses multiple type parameters. Example 5.5 gives the code.

5.5 GenericMethodDemo.cs

```
using System;

public class GenericMethodDemo {

  public T PrintValue<T, U>(T param1, U param2){
    T local_var = param1;
    Console.WriteLine("Parameter values are: param1 = " + param1 + " param2 = " + param2);
    Console.WriteLine("Local variable value is: local_var = " + local_var);
    return local_var;
  }
}
```


Referring to example 5.5 — The GenericMethodDemo class is an ordinary, non-generic class. On line 5 it defines a generic method named PrintValue<T, U> that uses two type parameters T and U in its definition. This example demonstrates how the type parameters T and U can be used to specify the return type, method parameters, or local variables within the method. Example 5.6 demonstrates the use of the generic PrintValue<T, U>() method.

5.6 MainApp.cs

```
1       using System;
2
3       public class MainApp {
4         public static void Main(){
5           GenericMethodDemo gmd = new GenericMethodDemo();
6           gmd.PrintValue<string, int>("Hello World", 4); // explicit type arguments
7           Console.WriteLine("--------------------------");
8           gmd.PrintValue(125.25, 62); //using generic type inference
9         }
10      }
```

Referring to example 5.6 — an instance of GenericMethodDemo is created on line 5. On line 6, the generic PrintValue<T, U>() method is called. Notice how the type arguments string and int are explicitly specified between the angle brackets "< >". An alternative way to call a generic method is to let the compiler sort out the types via *generic type inference*. This concept is discussed in the following section.

Generic Type Inference

Again referring to example 5.6 — line 8 demonstrates how the type arguments for the generic PrintValue<T, U>() can be sorted out automatically by the compiler via *generic type inference*. The types inferred by the call to the PrintValue<T, U>() method are double and int respectively. Figure 5-3 shows the results of running example 5.6.

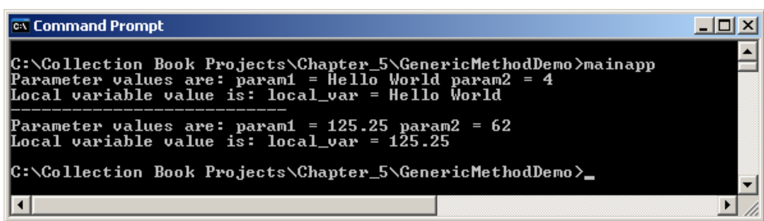


Figure 5-3: Results of Running Example 5.6

Quick Review

Generic methods use type parameters in their definition. A generic method definition may appear in the body of a non-generic class or structure.

There are two ways to call a generic method. 1) using explicit type arguments, or 2) letting the compiler figure out the types via generic type inference.

Generic Type Constraints

When defining generic types and methods using unbounded type parameters the compiler assumes the targeted base type is System.Object. This limits the range of valid operations you can perform on the subsequent type arguments you supply when you instantiate a generic type to those made public by the System.Object's interface. To overcome this limitation you must specify a *generic type constraint* that instructs the compiler to limit the range of authorized type arguments to those that subscribe to certain conditions.

There are six types of generic type constraints: 1) *default constructor constraint*, 2) *reference type constraint*, 3) *value type constraint*, 4) *class derivation constraint*, 5) *interface implementation constraint*, and 6) *naked constraint*. Of the six, you'll find numbers 4 and 5, the class derivation and interface implementation constraints, most useful for creating your own generic types. I discuss all six constraints in greater detail below.

Default Constructor Constraint

The default constructor constraint instructs the compiler to limit the range of acceptable type arguments to those types that supply a default constructor. A default constructor is a public constructor that omits a parameter list (i.e., a parameterless constructor).

At fist glance you may question the utility of this type of constraint, but if your generic type creates instances (objects) of the type arguments you supply, then those types will need to define a default constructor. A typical example of a class that creates objects is a *factory class*. Examples 5.7 through 5.9 give the code for a class named MyClass that implements a default constructor, a generic factory class named, appropriately enough, GenericFactory<T> where the type parameter T is constrained to types that implement a default constructor, and a class named MainApp that serves as a test driver.

5.7 MyClass.cs

```
public class MyClass {

  //field
  private string _field1;

  //default constructor
  public MyClass():this("Hello World"){ }

  //overloaded constructor
  public MyClass(string s){
    _field1 = s;
  }

  //property
  public string PropertyOne {
    get { return _field1; }
    set { _field1 = value; }
  }
}
```

Referring to example 5.7 — MyClass contains one field, two constructors, and a property named PropertyOne. One of the constructors is a default constructor. PropertyOne is a read-write property. Example 5.8 gives the code for the GenericFactory<T> class.

5.8 GenericFactory.cs

```
using System;

public class GenericFactory<T> where T: new() {

  private static GenericFactory<T> factory;

  public static GenericFactory<T> Instance {
    get { if (factory != null) {
            return factory;
          } else {
             factory = new GenericFactory<T>();
             return factory;
            }
        }
  }

  public T NewObject(){
    return new T();
  }
}
```

Referring to example 5.8 — the GenericFactory<T> class implements both the singleton and factory software design patterns. It also applies the *default constructor constraint* to the type parameter T. Notice how the constraint is defined. The keyword `where` is used followed by the parameter T, followed by a colon ':'. Following the colon the constraint `new()` signifies the default constructor constraint.

Here's how this class works. The GenericFactory<T> class defines one private static field named factory. It also defines one public static property named Instance. The Instance property is read-only. If the factory field is not null, meaning an instance of GenericFactory<T> has already been created, then a reference to the field is returned. If the factory field is null, a new instance of GenericFactory<T> is created and assigned to the factory field before it is returned.

The NewObject() method simply returns new references to objects of type T. Note that the default constructor is used to create objects of type T. (i.e., T()). Example 5.9 shows the GenericFactory<T> class in action.

5.9 MainApp.cs

```
using System;

public class MainApp {
  public static void Main(){
    MyClass mc1 = GenericFactory<MyClass>.Instance.NewObject();
    MyClass mc2 = GenericFactory<MyClass>.Instance.NewObject();
    Console.WriteLine(mc1.PropertyOne);
    Console.WriteLine(mc2.PropertyOne);
    Console.WriteLine("----------------------------");
    mc1.PropertyOne = "A slightly different message string...";
    Console.WriteLine(mc1.PropertyOne);
    Console.WriteLine(mc2.PropertyOne);
  }
}
```

Referring to example 5.9 — the GenericFactory<T> class is used to create instances of MyClass as is indicated by using the type MyClass as a type argument (i.e., GenericFactory<MyClass>). Since the Instance property is static, it's accessed via the type name. The type name in this case is GenericFactory<MyClass>. The NewObject() method, which is a non-static method, is called via the reference returned as a result of accessing the GenericFactory<MyClass>.Instance property. The remaining code in example 5.9 then exercises the two MyClass objects retrieved via the factory. Figure 5-4 shows the results of running this program.

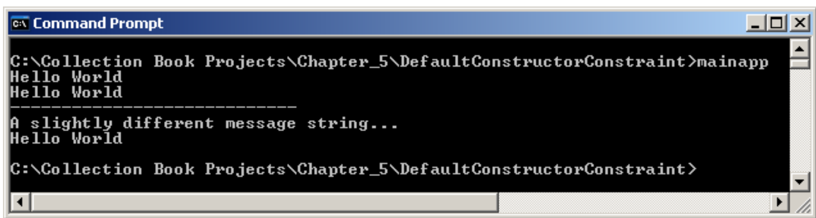


Figure 5-4: Results of Running Example 5.9

Note that the GenericFactory<T> class can be used to generate objects of any type that implements a default constructor. Remember that in C# if you don't explicitly define at least one constructor the compiler will supply a default constructor. This is good enough to satisfy the default constructor constraint. Example 5.10 gives the code for a simple class that leaves the generation of a default constructor up to the compiler. Example 5.11 then uses the GenericFactory<T> class to create an object of this type.

5.10 SimpleClass.cs

```
public class SimpleClass {

  private string _field1 = "Hello World";

  // default constructor generated by compiler

  public string PropertyOne {
    get {  return _field1; }
    set {  _field1 = value; }
  }
}
```

Referring to example 5.10 — the definition of SimpleClass leaves the generation of the default constructor to the compiler. It defines one read-write property named PropertyOne.

5.11 MainApp.cs

```
using System;

public class MainApp {
  public static void Main(){
    SimpleClass sc = GenericFactory<SimpleClass>.Instance.NewObject();
    Console.WriteLine(sc.PropertyOne);
  }
}
```

Figure 5-5 shows the results of running this program.

Reference/Value Type Constraints

The purpose of the reference and value type constraints is to limit the range of valid type arguments to either reference types (classes) or value types (structures). You would use either of these constraints when, in order for the generic code to work properly, it needs to know if it's dealing with reference types or with value types. An example of

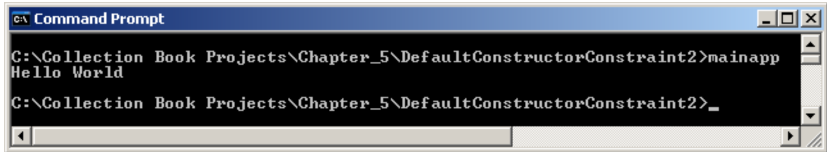


Figure 5-5: Results of Running Example 5.11

when this distinction would be important is when the semantics, or the meaning, of a particular expression would change according to whether the operands were classes or structures. (Classes have reference semantics and structures have value semantics.)

A case in point would be the difference in the behavior of *comparison or equality semantics* of reference types vs. value types. By default, reference types, when compared to each other for equality, perform a comparison (i.e., the Object.Equals() method) of one reference to another. Thus, two reference type objects with equal values will be found to be NOT equal if they are two distinct objects residing in two distinct memory locations. This is how comparisons work for reference types unless you explicitly override the Object.Equals() method to clarify the semantics of the equality comparison for your user-defined classes.

Conversely, a comparison of value type objects tests the contents of one against the contents of another, not their addresses.

Reference Type Constraint

Let's take a look first at an example of a reference type constraint. To help with this example I shall enlist the aid of a class named MyClass given in example 5.12.

5.12 MyClass.cs

```
public class MyClass {
  private int _field;

  public MyClass():this(0){ }

  public MyClass(int val){
    _field = val;
  }

  public int Value {
     get {  return _field; }
     set {  _field = value; }
  }
}
```

Referring to example 5.12 — MyClass defines one integer field named _field, two constructors, and one integer read-write property named Value. Fairly straight forward so far, nothing fancy. Example 5.13 gives the code for a generic class named EqualityChecker<T> where the type parameter T has been constrained to reference types.

5.13 EqualityChecker.cs

```
1       using System;
2
3       public class EqualityChecker<T> where T: class {
4
5          public bool CheckEquality(T a, T b){
6            bool result = a.Equals(b);
7            Console.WriteLine( result +  ": {0} is " + (result?"":"not ") + "equal to {1}", a, b);
8            return ( a.Equals(b));
9          }
10      }
```

Referring to example 5.13 — the EqualityChecker<T> class applies a reference type constraint on the type parameter T. Note how the constraint is applied with the use of the `class` keyword (where T : class). The EqualityChecker<T> class then goes on to define one method named CheckEquality() that takes two arguments of type T and performs an equality comparison of the two objects using the System.Equals() method on line 6. Note that the assumption made here is that all objects supplied, being class types, will subscribe to reference semantics, but this becomes hard to enforce since the Object.Equals() method can be overridden and thus its default behavior changed in derived classes. A case in point is the String class, where the System.Object() method is overridden to perform a comparison of one string's value (character sequence) against another's. Let's take a look at the EqualityChecker<T> class in action.

5.14 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        EqualityChecker<string> eq1 = new EqualityChecker<string>();
6        eq1.CheckEquality("Hello", "Hello");
7        eq1.CheckEquality("Hello", "World");
8        Console.WriteLine("------------------------------------");
9        EqualityChecker<MyClass> eq2 = new EqualityChecker<MyClass>();
10       eq2.CheckEquality(new MyClass(5), new MyClass(5));
11     }
12   }
```

Referring to example 5.14 — the EqualityChecker<T> class is instantiated first using the string type. On lines 6 and 7 the CheckEquality() method is called using string literals, first with two of the same value, next with two different values. On line 9 a new instance is EqualityChecker<T> is created using the MyClass type. On line 10 the CheckEquality() method is once again called with two instances of MyClass whose fields are initialized to the same value (5). Figure 5-6 shows the results of running this program.

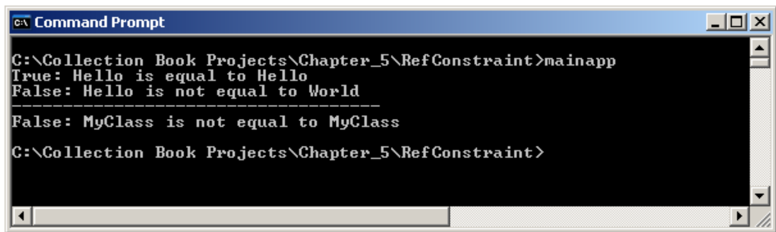


Figure 5-6: Results of Running Example 5.14

Referring to figure 5-6 — note the output for the string objects vs. the MyClass objects. Strings that contain identical character sequences are considered equal because that's the behavior defined by the String class's version of the overridden Object.Equals() method. Had I overridden the System.Equals() method in MyClass, I could have instructed it to behave in a similar fashion, comparing fields instead of addresses. But I didn't, and when writing generic code, you can't assume anything other that the worst case or common denominator.

Value Type Constraint

Let's look now at a modified version of the EqualityChecker<T> class, but this time I shall constrain the type parameters to value types. To help in this example I will use the MyStruct structure, the code for which is given in example 5.15.

5.15 MyStruct.cs

```
public struct MyStruct {

 private int _field;

 public MyStruct(int val){
   _field = val;
 }

 public int Value {
   get {  return _field; }
   set {  _field = value; }
 }

}
```

Referring to example 5.15 — the MyStruct structure defines one private integer field named _field, one constructor (explicit default constructors are not allowed in structures), and one public property named Value.

Example 5.16 gives the code for the slightly modified EqualityChecker<T> class that now has a *value type constraint* applied to the type parameter T.

5.16 EqualityChecker.cs (Mod 1)

```
using System;

public class EqualityChecker<T> where T: struct {

   public bool CheckEquality(T a, T b){
     bool result = a.Equals(b);
     Console.WriteLine( result +  ": {0} is " + (result?"":"not ") + "equal to {1}", a, b);
```



```
8            return ( a.Equals(b));
9          }
10       }
```

Referring to example 5.16 — with the exception of the value type constraint applied on line 3 using the `struct` keyword, this code remains otherwise unchanged from the previous version of EqualityChecker<T>. Example 5.17 puts this new version of EqualityChecker<T> through some paces.

5.17 MainApp.cs

```
1        using System;
2
3        public class MainApp {
4          public static void Main(){
5            EqualityChecker<int> eq1 = new EqualityChecker<int>();
6            eq1.CheckEquality(2, 2);
7            eq1.CheckEquality(3, 4);
8            Console.WriteLine("-------------------------");
9            EqualityChecker<MyStruct> eq2 = new EqualityChecker<MyStruct>();
10           eq2.CheckEquality(new MyStruct(5), new MyStruct(5));
11         }
12       }
```

Referring to example 5.17 — on line 5 an instance of EqualityChecker<T>, eq1, is instantiated using an integer type argument. Lines 6 and 7 demonstrate calls to the CheckEquality() method using various integer literal values. On line 9 a second instance of EqualityChecker<T> is created using the MyStruct value type. On line 10 the CheckEquality() method is called using two fresh instances of MyStruct initialized to hold the same values. Figure 5-7 shows the results of running this program.

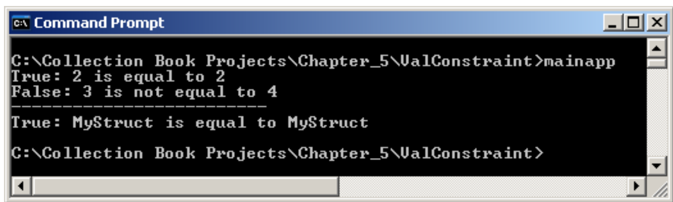


Figure 5-7: Results of Running Example 5.17

Class/Interface Derivation/Implementation Constraints

The class derivation and interface implementation constraints are by far the most useful generic type constraints. By targeting a specific type name, whether it be a class or an interface, you inform the compiler of your intent to use a specified set of operations on the type parameters. It is only by using the interface implementation or class derivation constraints that you can access overloaded operators. These constraints also let you take advantage of type substitution by specifying abstract base classes, interfaces, or a combination of both, and then substituting derived types in their place.

Interface Implementation Constraint

The interface implementation constraint lets you constrain type parameters to a specific interface type. Example 5.18 gives the code for a generic class named EqualityChecker<T> that constrains the type parameter T to objects that implement the IComparable and IComparable<T> interfaces.

5.18 EqualityChecker.cs

```
using System;

public class EqualityChecker<T> where T: IComparable, IComparable<T> {

   public bool CheckEquality(T a, T b){
     bool return_val = false ;
     int result = a.CompareTo(b);
     if(result == 0){
       return_val = true;
     }
     Console.WriteLine(return_val + ": {0} is " + (return_val?"":"not ") + "equal to {1}", a, b);
     return return_val;
   }
}
```


Referring to example 5.18 — the EqualityChecker<T> class specifies two interface names in a comma-separated constraint list. On line 5, the CheckEquality() method takes two arguments of type T and compares one against the other using the CompareTo() method. (Types that realize the IComparable or IComparable<T> interfaces implement the CompareTo() method.) Line 11 prints the results of the comparison to the console.

Example 5.19 provides a short program that shows the EqualityChecker<T> class in action.

5.19 MainApp.cs

```
1       using System;
2
3       public class MainApp {
4         public static void Main(){
5           EqualityChecker<string> eq1 = new EqualityChecker<string>();
6           eq1.CheckEquality("Hello", "Hello");
7           Console.WriteLine("-----------------------");
8           EqualityChecker<int> eq2 = new EqualityChecker<int>();
9           eq2.CheckEquality(2, 4);
10          Console.WriteLine("-----------------------");
11        }
12      }
```

Referring to example 5.19 — an instance of EqualityChecker<T> is created on line 5 using the string type. On line 6, the CheckEquality() method compares two string literal values. On line 8 another instance of Equality-Checker() is instantiated but this time using an integer type argument. (Any type will work so long as it implements either the IComparable or IComparable<T> interfaces.) Figure 5-8 shows the results of running this program.

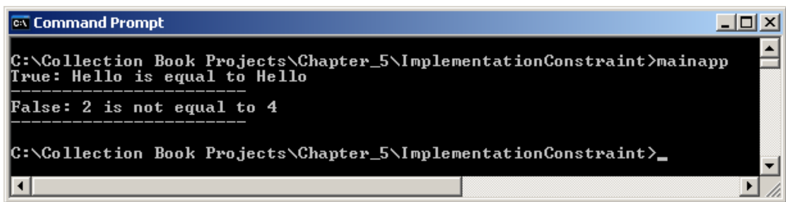


Figure 5-8: Results of Running Example 5.19

Class Derivation Constraint

The class derivation constraint lets you target a specific class type by name. Specifying a class name implies that subtypes of that class can be used as type arguments as well.

As I mentioned above, using either an interface implementation constraint or class derivation constraint is the only way to go if you wish to utilize overloaded operators on your type parameters. Example 5.20 gives the code for a class named MyType that overloads several operators. The MyType class is nothing more than a wrapper for an integer object.

5.20 MyType.cs

```
using System;

public class MyType {
  private int _intField;

  public int IntField {
    get {  return _intField; }
    set {  _intField = value; }
  }

  public MyType():this(5){
  }

  public MyType(int intField){
    _intField = intField;
  }

  public static MyType operator +(MyType mt){
    mt.IntField = (+mt.IntField);
    return mt;
  }

  public static MyType operator -(MyType mt){
    mt.IntField = (-mt.IntField);
    return mt;
  }
```



```

        public static bool operator ! (MyType mt){
          bool retVal = true;
          if(mt.IntField >= 0){
            retVal = false;
          }
          return retVal;
        }

        public static bool operator true(MyType mt){
          return !mt;
        }

        public static bool operator false(MyType mt){
         return !mt;
        }

        public static MyType operator ++ (MyType mt){
          MyType result = new MyType(mt.IntField);
          ++result.IntField;
          return result;
        }

        public static MyType operator -- (MyType mt){
          MyType result = new MyType(mt.IntField);
          --result.IntField;
          return result;
        }

        public static MyType operator +(MyType lhs, MyType rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField += rhs.IntField;
          return result;

        }

        public static MyType operator -(MyType lhs, MyType rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField -= rhs.IntField;
          return result;
        }

        public static MyType operator +(MyType lhs, int rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField += rhs;
          return result;
        }

        public static MyType operator -(MyType lhs, int rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField -= rhs;
          return result;
        }

        public static MyType operator *(MyType lhs, MyType rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField *= rhs.IntField;
          return result;
        }

        public static MyType operator *(MyType lhs, int rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField *= rhs;
          return result;
        }

        public static MyType operator /(MyType lhs, MyType rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField /= rhs.IntField;
          return result;
        }

        public static MyType operator /(MyType lhs, int rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField /= rhs;
          return result;
        }

        public static MyType operator &(MyType lhs, MyType rhs){
          MyType result = new MyType(lhs.IntField);
          result.IntField &= rhs.IntField;
```

```
    return result;
  }

  public static MyType operator |(MyType lhs, MyType rhs){
    MyType result = new MyType(lhs.IntField);
    result.IntField |= rhs.IntField;
    return result;
  }

  public static bool operator ==(MyType lhs, MyType rhs){
    return lhs.IntField == rhs.IntField;
  }

  public static bool operator !=(MyType lhs, MyType rhs){
    return lhs.IntField != rhs.IntField;
  }

  public static bool operator <(MyType lhs, MyType rhs){
    return lhs.IntField < rhs.IntField;
  }

  public static bool operator >(MyType lhs, MyType rhs){
    return lhs.IntField > rhs.IntField;
  }

  public static bool operator <=(MyType lhs, MyType rhs){
    return lhs.IntField <= rhs.IntField;
  }

  public static bool operator >=(MyType lhs, MyType rhs){
    return lhs.IntField >= rhs.IntField;
  }

  public static explicit operator int(MyType mt){
    return mt.IntField;
  }


  // overridden System.Object methods
  public override String ToString(){
    return IntField.ToString();
  }

  public override bool Equals(object o){
    if(o == null) return false;
    if(!(o is MyType)) return false;
    return this.ToString().Equals(o.ToString());
  }

  public override int GetHashCode(){
    return this.ToString().GetHashCode();
  }
} // end class definition
```

Referring to example 5.20 — the MyType class overloads most of the important operators. If you're unfamiliar with operator overloading please refer to chapter 22 in my book *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming* (ISBN(13) 978-1-932504-07-1).

Example 5.21 gives the code for a generic class named, simply enough, GenericType<T>. (I must be running out of steam here!)

5.21 GenericType.cs

```
using System;

public class GenericType<T> where T: MyType {

  public void PrintSum(T arg1, T arg2){
    Console.WriteLine(arg1 + " + " + arg2 + " = " + (arg1 + arg2));
  }
}
```

Referring to example 5.21 — The GenericType<T> class constrains the type parameter T to objects of type MyType or its subtypes. It defines one method named PrintSum() that takes two arguments of type T which, because of the derivation constraint, are guaranteed to be objects of type MyType. It then applies the binary addition operator '+' and prints the sum of the two objects to the console. Figure 5-9 shows the results of running this program.

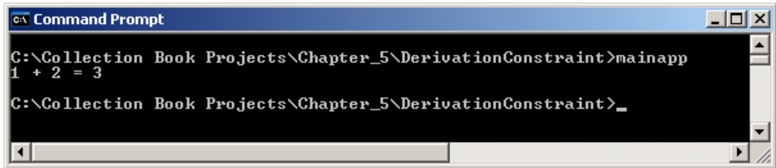


Figure 5-9: Results of Running Example 5.21

Naked Constraint

The naked constraint is used to define one type parameter in terms of another. To demonstrate the naked constraint I'll use two classes, one named BaseClass and one named DerivedClass, which, as you'll see, derives from BaseClass. Example 5.22 gives the code for BaseClass.

5.22 BaseClass.cs

```
1        public class BaseClass {
2
3          public virtual string  InterfaceMethod(){
4            return "String returned from BaseClass";
5          }
6        }
```

Referring to example 5.22 — BaseClass defines one public virtual method named InterfaceMethod(). All this method does when called is return the string literal shown on line 4. Example 5.23 gives the code for DerivedClass.

5.23 DerivedClass.cs

```
1        public class DerivedClass : BaseClass {
2
3           public override string InterfaceMethod(){
4             return "String returned from Derived class method.";
5           }
6        }
```

Referring to example 5.23 — DerivedClass extends BaseClass and provides its own implementation of InterfaceMethod() by overriding the BaseClass.InterfaceMethod(). Example 5.24 gives the code for a class named GenericClass<T>.

5.24 GenericClass.cs

```
1        using System;
2
3        public class GenericClass<T> where T: BaseClass {
4
5          public void GenericMethod<U>(U arg) where U: T {
6            Console.WriteLine(arg.InterfaceMethod());
7          }
8        }
```

Referring to example 5.24 — the GenericClass<T> applies a derivation constraint to the type parameter T limiting the range of acceptable type arguments to type BaseClass and its derived types. On line 5 a generic method named GenericMethod<U>(U arg) is defined and a naked constraint is applied to the type parameter U that says, in effect, "limit U to the type specified by T or its subtypes." Example 5.25 shows the GenericClass<T> in action.

5.25 MainApp.cs

```
1        using System;
2
3        public class MainApp {
4          public static void Main(){
5            GenericClass<BaseClass> gc1 = new GenericClass<BaseClass>();
6            gc1.GenericMethod<BaseClass>(new BaseClass());
7            gc1.GenericMethod<BaseClass>(new DerivedClass());
8            gc1.GenericMethod<DerivedClass>(new DerivedClass());
9          }
10       }
```

Referring to example 5.25 — an instance of GenericClass<T> is created on line 5 using BaseClass as a type argument. Notice on lines 6 through 8 how three different versions of the GenericMethod<U> method are called. On line 6, BaseClass is used a a type argument and a new instance of BaseClass is created and used as a method argument; on line 7, BaseClass is used as a type argument and a new instance of DerivedClass is created and used as a method argument; and finally, on line 8. DerivedClass is used as a type argument and a new instance of DerivedType is created and used as a method argument. Figure 5-10 shows the results of running this program.

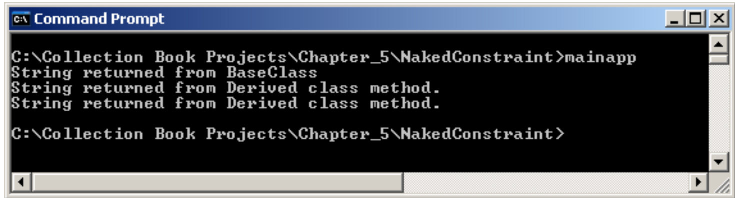


Figure 5-10: Results of Running Example 5.25

Limited Utility of the Naked Constraint

In example 5.24 I combined the *derivation constraint* with the *naked constraint* for demonstration purposes. Had I not specified the derivation constraint limiting the acceptable type arguments to those of type BaseClass and its derivatives, I would have been unable to access the InterfaceMethod() method in the code and would have been limited to the interface published by the System.Object class. With this in mind, I could have simply done away with the naked constraint as applied to the GenericMethod<U>() method and rewritten the GenericClass<T> with only the derivation constraint to the same effect. The simplified code for an alternative version of GenericClass<T> appears in example 5.26.

5.26 GenericClass.cs (Mod 1)

```
using System;

public class GenericClass<T> where T: BaseClass {

  public void GenericMethod(T arg){
    Console.WriteLine(arg.InterfaceMethod());
  }
}
```

Referring to example 5.26 — the naked constraint has been removed for the GenericMethod() declaration. This simplifies the code with no effect on functionality. The derivation constraint limits the range of type arguments to BaseClass and its derived types, but as you have learned, the derivation and interface constraints are the most useful to you anyway.

Example 5.27 gives the modified MainApp class.

5.27 MainApp.cs (Mod 1)

```
using System;

public class MainApp {
  public static void Main(){
    GenericClass<BaseClass> gc1 = new GenericClass<BaseClass>();
    gc1.GenericMethod(new BaseClass());
    gc1.GenericMethod(new DerivedClass());
  }
}
```

Figure 5-11 shows the results of running this program.

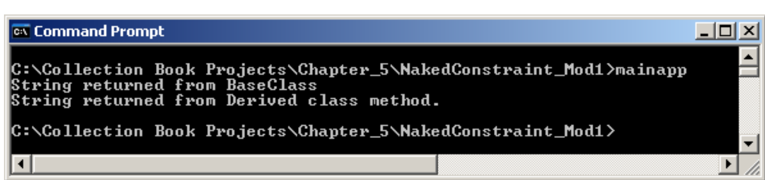


Figure 5-11: Results of Running Example 5.27

Constraint Summary Table

Table 5-1 lists and summarizes the constraints presented in this section. It offers recommendations for their use and briefly describes issues you need to consider when applying constraints.

Constraint	Form	Implementation
Default Constructor Constraint	<T> where T: new() { ... }	Use when code needs to create objects of type T
Reference Type Constraint	<T> where T: class { ... }	Use when code needs to know if reference semantics apply to objects of type T.
Value Type Constraint	<T> where T: struct { ... }	Use when code needs to know if value type semantics apply to objects of type T.
Interface Implementation Constraint	<T> where T: *interface_name* { ... }	Use when code needs to know that objects of type T implement the interface as indicated by *interface_name*. This is a very useful constraint because it lets you access all operations defined by the specified interface.
Class Derivation Constraint	<T> where T: *class_name* { ... }	Use when code needs to know that objects of type T are derrived from the class indicated by *class_name*. This is a very useful constraint because it lets you access all operations defined by the specified class interface.
Naked Constraint	<T, U> where T: U { ... }	Specifies objects of type T in terms of U. Effectively limits objects of T to those of type U and its derivatives. Limited usefulness because the only operations available on objects of type T are those specified by System.Object. Favor the use of either the derivation constraint or implementation constraint.

Table 5-1: Constraint Summary Table

Quick Review

Use generic type constraints to limit the range of acceptable type arguments in generic types. There are six type constraints: 1. Default constructor constraint, 2. Reference type constraint, 3. Value type constraint, 4. Interface implementation constraint, 5. Class derivation constraint, and 6. Naked constraint. Of the six, the interface implementation and class derivation constraints are most helpful.

Benefits of Using Generic Types

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality. I discuss each of these benefits in more detail below.

Increased Type Safety

The use of generics reduces and in many cases eliminates the need for the programmer to perform type checks and casting operations. When you create a generic type, type checks on the type parameters and type arguments are performed at compile time, eliminating runtime type errors.

As you learned in this chapter, the enforcement of type safety imposes limits on what you can get away with when you create a generic type. For example, you can't apply operators willy-nilly to unbounded type parameters because the compiler can't guarantee the type argument eventually supplied to instantiate the generic type will implement those operators.

Generics Save Space

The key rationale for generic types derives from the benefit of writing a *general-purpose routine* that can be used in *multiple contexts*. This saves space because it eliminates the need for multiple code assemblies, each one perhaps created to manipulate different types of objects using the same, repeated code pattern.

Generics Improve Performance

Generics types have the potential to improve code performance, especially in compute-intensive code segments where the boxing and unboxing of value-types would incur overhead. To illustrate this point I have written a short program that adds 1 million integers to a non-generic ArrayList and a generic List<int> and then sorts each list, recording the time it takes to complete the sort operation on each collection. Example 5.28 gives the code.

5.28 PerformanceTest.cs

```
using System;
using System.Collections;
using System.Collections.Generic;

public class PerformanceTestOne {
  public static void Main(){
    ArrayList list = new ArrayList();
    List<int> generic_list = new List<int>();
    int NUMBER = 1000000;

    Console.WriteLine("Adding {0:N0} integers to lists", NUMBER);
    Console.WriteLine("--------------------------------");
    Random random = new Random();
    for(int i=0; i<NUMBER; i++){
      int temp = random.Next();
      list.Add(temp);
      generic_list.Add(temp);

    }
    DateTime start = DateTime.Now;
    Console.WriteLine("Sorting ArrayList -> Start time: " + start);
    list.Sort();
    TimeSpan array_list_elapsed_time = (DateTime.Now - start);
    Console.WriteLine("ArrayList sorted in: " + array_list_elapsed_time);

    Console.WriteLine("-----------------------------------");

    start = DateTime.Now;
    Console.WriteLine("Sorting List<int> -> Start time: " + start);
    generic_list.Sort();
    TimeSpan list_elapsed_time = (DateTime.Now - start);
    Console.WriteLine("ArrayList sorted in: " + list_elapsed_time);

    Console.WriteLine("-----------------------------------");
    Console.WriteLine("Time difference: " + (array_list_elapsed_time - list_elapsed_time));
  }
 }
```

Referring to example 5.28 — this program compares the performance of a non-generic collection against that of a generic collection in the sorting of integers. The non-generic ArrayList will incur a boxing and unboxing performance penalty because integer value-types must be "boxed" into objects when being inserted into the ArrayList, (since it is object-based) and unboxed when performing the sort comparisons. Figure 5-12 shows the results of running this program.

Referring to figure 5-12 — the old-school ArrayList collection took 1.98 seconds to sort while the generic List<int> collection took only .218 seconds to sort. That's an improvement of approximately 90%. Your times will most certainly vary from mine but you should see similar results.

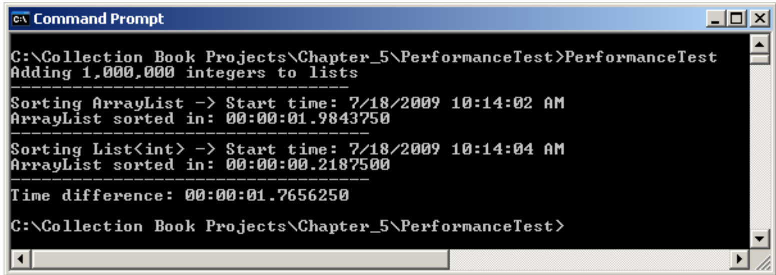


Figure 5-12: Results of Running Example 5.28

Generics Eliminate Work and Improve Code Quality

The use of generic types, that is, the creation of general purpose code that works with multiple data types, can potentially save you a lot of work and improve code quality. You save time and eliminate redundant work by writing code that can be reused in different contexts. For example, code that sorts Strings can sort numeric data as well. (The ordering behavior is implemented in the targeted data type, as you'll learn in subsequent chapters.)

Code that can be reused in different contexts tends to have more of its bugs worked out. This can really be applied not only to generics, but to the whole .NET framework. The .NET framework is not without its issues and problems, but the more the code is used and tested, the more bugs are discovered and fixed in subsequent releases.

Quick Review

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality.

Summary

A *generic type* is one that's declared with the help of one or more *type parameters*. A type parameter serves as a place holder which will eventually be replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a type *template*; the purpose of the template is to create new types as if stamping them from a mold.

You can create generic types that use one or more type parameters. Type parameters can appear in the definition of any type member, including fields, constructors, properties, methods, etc.

In the absence of a type parameter *constraint*, the compiler assumes the targeted interface will be that of the System.Object class. An unconstrained type parameter is referred to as an *unbounded type parameter*.

Generic methods use type parameters in their definition. A generic method definition may appear in the body of a non-generic class or structure. There are two ways to call a generic method. 1) using explicit type arguments, or 2) letting the compiler figure out the types via generic type inference.

Use generic type constraints to limit the range of acceptable type arguments in generic types. There are six type constraints: 1. *Default constructor constraint*, 2. *Reference type constraint*, 3. *Value type constraint*, 4. *Interface implementation constraint*, 5. *Class derivation constraint*, and 6. *Naked constraint*. Of the six, the interface implementation and class derivation constraints are most helpful.

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality.

References

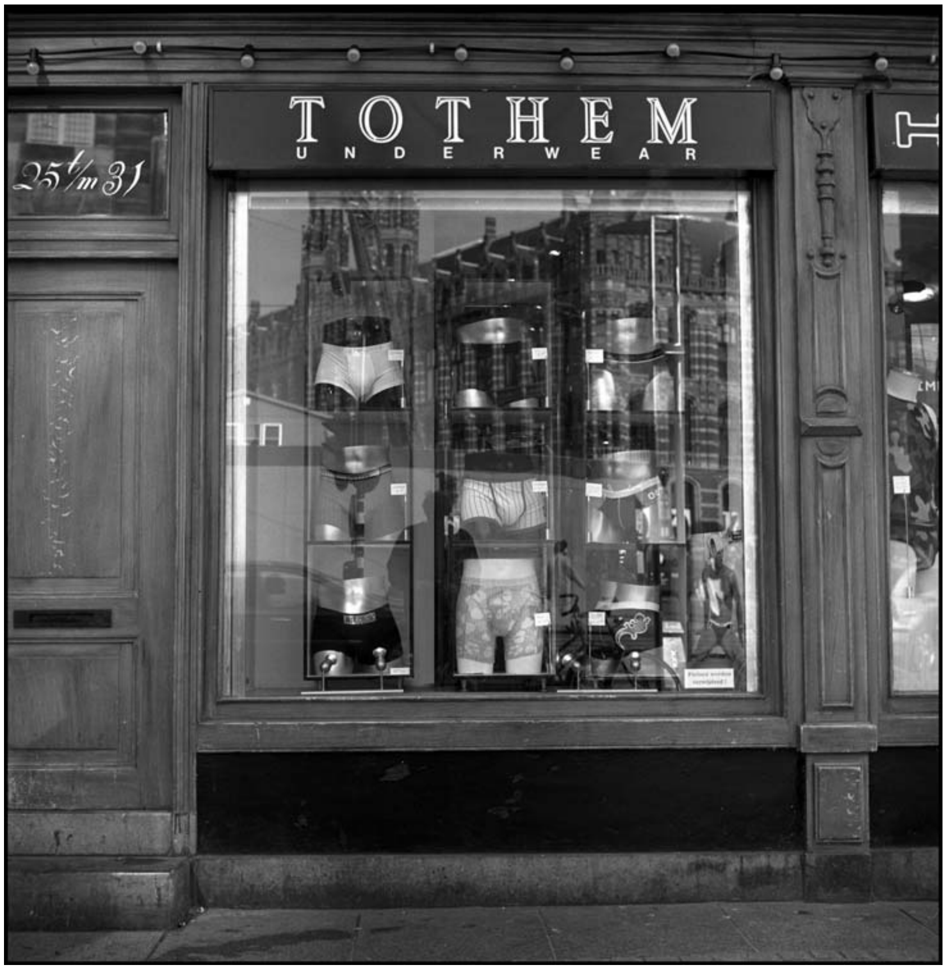
ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

Chapter 6



Yashica Mat 124G

Underwear Window

Lists

Learning Objectives

- Describe the features of the ArrayList class
- Describe the features of the List<T> class
- Describe the features of the LinkedList<T> class
- Add elements to a list using the Add() method
- Access individual elements of a list using array indexer notation
- Apply casting to objects retrieved from an ArrayList
- Use the façade software design pattern to make an ArrayList type safe
- Sort the contents of a list using the natural ordering of contained objects
- Reverse the contents of a list
- State the functionality provided by the IList, ICollection, and IEnumerable interfaces
- State the functionality provided by the IList<T>, ICollection<T>, and IEnumerable<T> interfaces

Introduction

Lists are perhaps the most often used collection types of which there are two primary varieties: array-based lists, and linked lists.

As their name implies, lists store their contents in a sequentially accessible fashion, but there's a big difference between the behavior of an array-based list and a linked list. In this chapter I will explain the difference between these two list types and give you a glimpse into the inner workings of each.

I've already introduced you to the ArrayList and its generic relative, List<T>, in chapter 1. It's my intention here to dive deeper into the operations of lists and expand your repertoire by introducing you to the generic LinkedList<T> collection.

Many times I am asked, "How do array-based lists expand automatically to hold additional elements?" I will answer this question with a short demo program that shows how an array-based collection dynamically resizes itself to enlarge its capacity. I will also show you how linked lists operate, complete with sample code showing how individual list nodes are inserted and deleted, and how items on the list are located. In this regard, this chapter doubles as a short course in data structures.

Armed with a deeper understanding of these concepts, you'll be better able to select the collection class most appropriate to your particular programming situation.

Array-Based List Collections – How They Work

Arrays serve as the foundational data structure for array-based collections. In this section I will show you how an array-based collection automatically grows to accommodate the addition or insertion of more objects than its initial capacity allows. This dynamic growth capability is a primary performance characteristic of array-based lists that you must take into account when you use one in your code, especially if you plan to manipulate lists with large numbers of elements. Let's look at a home-grown, array-based collection class to demonstrate the dynamic resizing capability.

A Home-Grown Dynamic Array

Imagine for a moment that you are working on a project and you're deep into the code. You're in the flow and you don't want to stop to read no stinkin' API documentation. The problem at hand dictates the need for an array with special powers — one that can automatically grow itself when one too many elements are inserted. To solve your problem, you hastily crank out the code for a class named DynamicArray shown in example 6.1.

6.1 DynamicArray.cs

```
using System;

public class DynamicArray {
  private Object[]  _object_array = null;
  private int _next_open_element = 0;
  private int _growth_increment = 10;
  private const int INITIAL_SIZE = 25;

  public int Count {
     get {  return _next_open_element; }
  }

  public object this[ int index] {
     get {
       if((index >= 0) && (index <  _object_array.Length)){
          return _object_array[ index] ;
       } else throw new ArgumentOutOfRangeException();
     }
     set {
      if(_next_open_element <  _object_array.Length){
             _object_array[ _next_open_element++]  = value;
          } else{
        GrowArray();
        _object_array[ _next_open_element++]  = value;
      }
     }
```



```
27     }
28
29    public DynamicArray(int size){
30      _object_array = new Object[ size] ;
31    }
32
33    public DynamicArray():this(INITIAL_SIZE){ }
34
35    public void Add(Object o){
36      if(_next_open_element < _object_array.Length){
37         _object_array[ _next_open_element++]  = o;
38      } else{
39     GrowArray();
40     _object_array[ _next_open_element++]  = o;
41       }
42    }  // end add() method;
43
44    private void GrowArray(){
45      Object[ ]  temp_array = _object_array;
46      _object_array = new Object[ _object_array.Length + _growth_increment] ;
47      for(int i=0, j=0; i<temp_array.Length; i++){
48        if(temp_array[ i]  != null){
49          _object_array[ j++]  = temp_array[ i] ;
50         }
51         _next_open_element = j;
52       }
53       temp_array = null;
54    }  // end growArray() method
55  }  // end DynamicArray class definition
```

Referring to example 6.1 — the data structure used as the basis for the DynamicArray class is an ordinary array of objects. Its initial size can be set via a constructor. If the default constructor is called, its initial size is set to the default value of 25 elements. Its growth increment is set to10 elements. This means that when the time comes to grow the array it will expand by 10 elements. In addition to its two constructors, the DynamicArray class has one property named Count, two additional methods named Add() and GrowArray(), and a class indexer member that starts on line 13. An indexer is a member that allows an object to be indexed using array notation.

The Add() method inserts an object reference into the next available array element pointed to by the _next_open_element variable. If the array is full, the GrowArray() method is called to grow the array. The GrowArray() method creates a temporary array of objects and copies each element to the temporary array. It then creates a new larger array and copies the elements into it from the temporary array.

The indexer member whose definition begins on line 13 allows you to access each element of the array. (**Note:** The array itself is private and therefore encapsulated, thus the need for the public indexer member to control access to the array.) If the index argument falls out of bounds, the indexer throws an ArgumentOutOfRangeException. The Count property simply returns the number of elements (references) contained in the array, which is the value of the _next_open_element variable. Example 6.2 shows the DynamicArray class in action.

6.2 ArrayTestApp.cs

```
1   using System;
2
3   public class ArrayTestApp {
4     public static void Main(){
5       DynamicArray da = new DynamicArray();
6       Console.WriteLine("The array contains " + da.Count + " objects.");
7       da.Add("Ohhh if you loved C# like I love C#!!");
8       Console.WriteLine(da[ 0] .ToString());
9       for(int i = 1; i<26; i++){
10        da.Add(i);
11      }
12      Console.WriteLine("The array contains " + da.Count + " objects.");
13      for(int i=0; i<da.Count; i++){
14        if(da[ i]  != null){
15           Console.Write(da[ i] .ToString() + ", ");
16          if((i%20)==0){
17            Console.WriteLine();
18           }
19        }
20      }
21      Console.WriteLine();
22    } //end Main() method
23  } // end ArrayTestApp class definition
```

Referring to example 6.2 — on line 5 an instance of DynamicArray is created using the default constructor. This results in an initial internal array length of 25 elements. Initially, its Count is zero because no references have yet been inserted. On line 7 a string object is added to the array and then printed to the console on line 8. The `for` state-

ment on line 9 inserts enough integers to test the array's growth capabilities. The `for` statement on line 13 prints all the non-null elements to the console. Figure 6-1 shows the results of running this program.

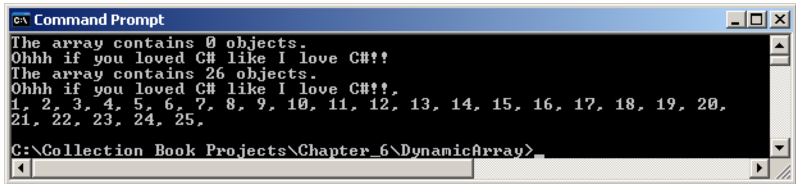


Figure 6-1: Results of Testing DynamicArray

Evaluating DynamicArray

The DynamicArray class works well enough for your immediate needs but it suffers several shortcomings that will cause serious problems should you try to use it in more demanding situations. For example, although you can access each element of the array, you cannot remove elements. You could add a method called Remove(), but what happens when the number of remaining elements falls below a certain threshold? You might want to shrink the array as well.

Another point to consider is how to insert references into specific element locations. When this happens, you must make room for the reference at the specified array index location and shift the remaining elements to the right. If you plan to frequently insert elements into your custom-built DynamicArray class, you will have a performance issue on your hands you did not foresee.

At this point, you would be well served to take a break from coding and dive into the API documentation to study up on the collections framework. There you will find that all this work, and more, is already done for you!

The ArrayList Class To The Rescue

Let's re-write the ArrayTestApp program with the help of the ArrayList class, which belongs to the .NET collections framework. Example 6.3 gives the code.

6.3 ArrayTestApp.cs (Mod 1)

```
1    using System;
2    using System.Collections;
3
4    public class ArrayTestApp {
5      public static void Main(){
6        ArrayList da = new ArrayList();
7        Console.WriteLine("The array contains " + da.Count + " objects.");
8        da.Add("Ohhh if you loved C# like I love C#!!");
9        Console.WriteLine(da[0].ToString());
10       for(int i = 1; i<26; i++){
11         da.Add(i);
12       }
13       Console.WriteLine("The array contains " + da.Count + " objects.");
14       for(int i=0; i<da.Count; i++){
15         if(da[i] != null){
16            Console.Write(da[i].ToString() + ", ");
17           if((i%20)==0){
18             Console.WriteLine();
19            }
20          }
21       }
22       Console.WriteLine();
23     }//end Main() method
24   }// end ArrayTestApp class definition
```

Referring to example 6.3 — I made only three changes to the original ArrayTestApp program: 1) I added another `using` directive on line 2 to provide access to the System.Collections namespace, 2) I changed the `da` reference declared on line 6 from a DynamicArray type to an ArrayList type, and 3) also on line 6, I created an instance of ArrayList instead of an instance of DynamicArray. Figure 6-2 shows the results of running this program.

If you compare figures 6-1 and 6-2 you will see that the output produced with an ArrayList is exactly the same as that produced using the DynamicArray. However, the ArrayList class provides much more ready-made functionality.

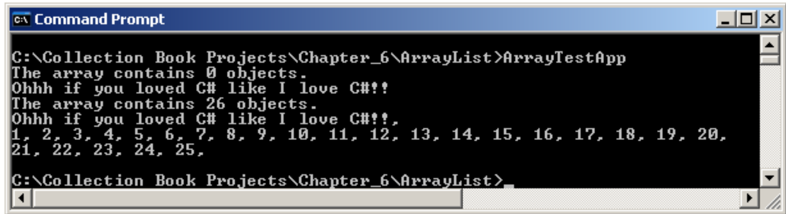


Figure 6-2: Results of Running Example 6.3

You might be asking yourself, "Why does this code work?" As it turns out, I gamed the system just a little bit. The DynamicArray class presented in example 6.1 just happens to partially and informally implement the IList interface. In other words, my DynamicArray class defines a subset of the properties and methods defined by the IList interface, which is implemented by the ArrayList class. Later, in example 6.3, when I changed the type of the `da` reference from DynamicArray to ArrayList and used the ArrayList collection class, everything worked fine.

Quick Review

Array-based lists, as their name implies, feature arrays as their fundamental data structure. Array-based lists are created with an initial capacity and can grow in size automatically to accommodate additional elements.

The Non-Generic ArrayList: Objects In – Objects Out

In this section I want to dive a bit deeper into the operation of an ArrayList collection class. I'll start with a discussion of the ArrayList's inheritance hierarchy and explain the functionality provided by each implemented interface. Following that, I'll show you how to provide a measure of type safety when using an ArrayList collection through the use of the façade software pattern.

ArrayList Inheritance Hierarchy

Figure 6-3 offers a class diagram showing the inheritance hierarchy of the ArrayList collection class.

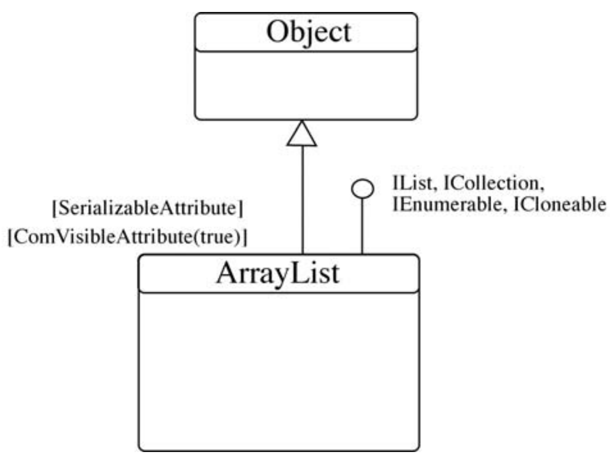


Figure 6-3: ArrayList Inheritance Hierarchy

Referring to figure 6-3 — the ArrayList class implicitly extends the Object class and implements the following interfaces: IList, ICollection, IEnumerable, and ICloneable. Additionally, the ArrayList class is tagged with two attributes: SerializableAttribute and ComVisibleAttribute(true).

An interface in C# may extend another interface and it will be helpful here to see the inheritance diagram for the ArrayList class drawn another way. Figure 6-4 offers an alternative UML class diagram of the ArrayList class.

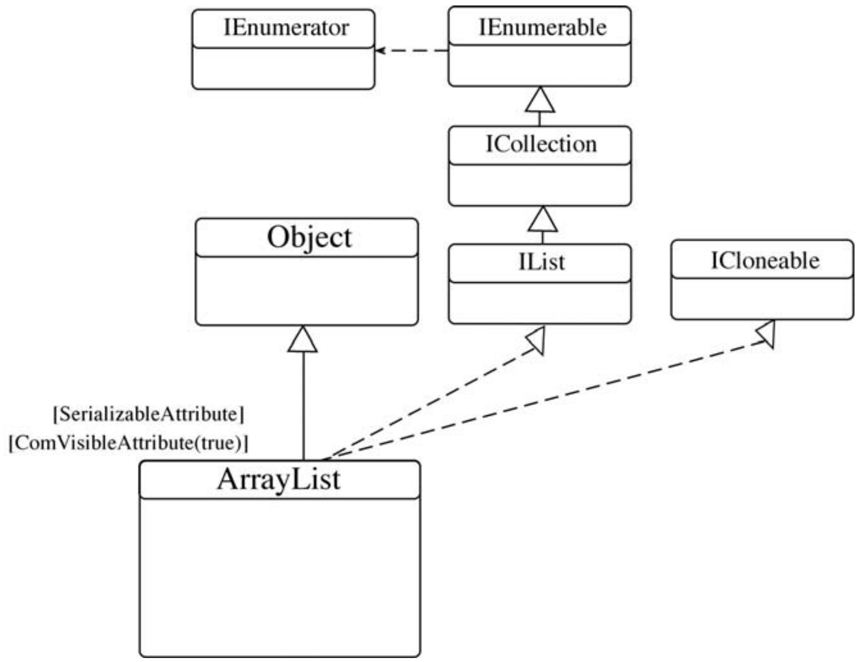


Figure 6-4: Expanded ArrayList Inheritance Hierarchy

Referring to figure 6-4 — the IList interface extends the ICollection interface, which in turn extends IEnumerable. Thus, any class that implements the IList interface is also implementing ICollection and IEnumerable. The IEnumerable interface has a dependency on the IEnumerator interface. The following sections explain the functionality provided by each of these interfaces and attributes.

Functionality Provided by the IEnumerable and IEnumerator Interfaces

Together, the IEnumerable and IEnumerator interfaces implement the *iterator* software design pattern in the .NET Framework. (See Eric Gamma, et. al., in the References section.) An iterator is an object that enables the standardized sequential traversal of the contents of a collection regardless of that collection's underlying structure. In other words, the iterator software pattern allows you to write polymorphic code that can step through the elements of a collection without you needing to worry about the messy details of how the collection is actually organized.

The IEnumerable interface declares one method named GetEnumerator() which returns the IEnumerator object for that particular collection. The IEnumerator object is then used to traverse the collection in a particular direction, with forwards, beginning with the first element, being the default implementation. However, you don't use the IEnumerator object directly; it's meant to be used with the `foreach` statement.

Collection classes are free to overload the GetEnumerator() method to provide additional ways of traversing the list. The ArrayList collection provides the GetEnumerator(int, int) method which allows the traversal of a range of elements contained in the list. When manually interacting with an IEnumerator object, you have access to two methods: MoveNext() and Reset(), and one property: Current.

When traversing a list, or any collection, with the help of an iterator, you generally say that you're "iterating over the collection." For example, if a colleague were to stop by your office and find you writing a `foreach` statement, and he asked you what on earth you were doing, you'd reply, "Why, I'm iterating over a collection!"

Example 6.4 demonstrates three ways of iterating over a list.

6.4 EnumeratorDemo.cs

```
using System;
using System.Collections;

public class EnumeratorDemo {

  public static void Main(){
    ArrayList list = new ArrayList();
    list.Add(1);
```



```
9        list.Add(2);
10       list.Add(3);
11       list.Add(4);
12
13       // Iterating over the list in the manner of old habits
14       for(int i = 0; i<list.Count; i++){
15         Console.Write(i + " ");
16       }
17       Console.WriteLine();
18
19       // Iterating over the list whilst ignoring the messy details
20       foreach(int i in list){
21         Console.Write(i + " ");
22       }
23       Console.WriteLine();
24
25       // Iterating over a list segment using overloaded GetEnumerator( ) method
26       // and directly manipulating the IEnumerator object via  IEnumerator.MoveNext()
27       IEnumerator e = list.GetEnumerator(1, 2);
28       while(e.MoveNext()){
29         Console.Write(e.Current + " ");
30       }
31     }
32   }
```

Referring to example 6.4 — a reference to an ArrayList is declared and initialized on line 7, followed by four consecutive calls to its Add() method. On line 14 a traditional `for` loop is used to iterate over the list and write each element to the console. I call this method "iterating over the list in the manner of old habits!" And quite frankly, when it comes to lists, it's a hard habit to break, but it does have its advantages, as you'll see later.

On line 19 I use the `foreach` statement to iterate over the list elements. The `foreach` uses the default implementation of the GetEnumerator() method, which allows the traversal of the collection in the forward direction. The only way to use the overloaded GetEnumerator(int, int) method to traverse the list is to directly manipulate the IEnumerator object using the MoveNext() method and accessing each element via the Current property as is demonstrated beginning on line 27.

Figure 6-5 shows the results of running example 6.4.

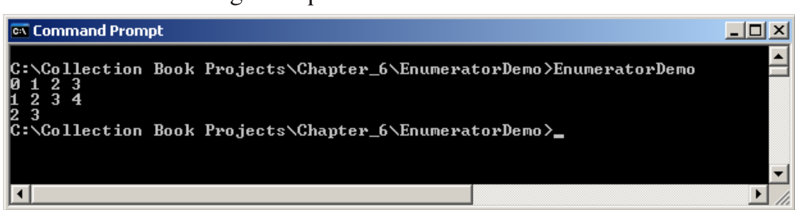


Figure 6-5: Results of Running Example 6.4

Collection Elements Cannot Be Modified When Using An Enumerator

Referring again to example 6.4 — the critically important difference between using the `for` loop vs. the `foreach` statement to iterate over the list is that you can modify the list elements in the body of the `for` loop but not in the body of the `foreach` loop nor in the body of the `while` loop when directly manipulating the enumerator. **The use of an enumerator to iterate over a collection results in a read-only sequence of elements.** Any attempt to modify the collection's elements, either by you or by another thread of execution, while stepping through the collection's elements with the enumerator, invalidates the enumerator and will result in an InvalidOperationException.

Where To Go From Here

For a detailed treatment of custom implementation of the IEnumerable and IEnumerator interfaces, Iterators, Iterator blocks, and named Iterators, please refer to Chapter 18: Creating Custom Collections. Collections and thread safety is covered in Chapter 14: Collections and Threads.

Functionality Provided by the ICollection Interface

The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. (i.e., All collections classes found in the System.Collections namespace.)

In addition to those methods declared by the IEnumerable interface, the ICollection interface provides the CopyTo() method which is used to copy the elements of the collection to a single-dimensional array, a feature you'll find to be quite handy on occasion. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used to coordinate multithreaded access to the collection and are discussed in detail in Chapter 14: Collections and Threads.

Functionality Provided by the IList Interface

The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection by an index, just like an array. In fact, as you learned in Chapter 3, arrays in the .NET Framework implement the IList and IList<T> interfaces. (**Note:** The indexer is listed in the Properties section of the MSDN documentation as Item.)

Functionality Provided by the ICloneable Interface

The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection. If you're creating a custom collection and you intend to implement the ICloneable interface, you'll need to be aware of the differences between a shallow copy and a deep copy. For a detailed discussion of deep copy vs. shallow copy please refer to my book *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming, Chapter 22: Well Behaved Objects* .

Functionality Provided by the SerializableAttribute

The SerializableAttribute informs the .NET runtime that the collection can be serialized. To serialize something means to convert it into a form that can be transmitted across a network or persisted to disk. Serialization is discussed in detail in Chapter 17: Collections and I/O.

The important thing to know about serialization at this point is that all objects contained within an ArrayList must be tagged with the Serializable attribute in order for a serialization operation on the list to succeed. If the list contains only types found in the .NET Framework then you're safe; it's with custom data types you need most concern yourself.

Functionality Provided by the ComVisibleAttribute(true)

The ComVisibleAttribute is used to control the visibility of a class and its members to the Component Object Model (COM). By default, all public types and their public members are visible to COM. I do not cover COM programming in this book so that's the last you'll hear about COM. If you would like to learn more about COM programming, I recommend the excellent book *Essential COM* by Don Box. (See the References section.)

Extension Methods

C# 3.0 introduced numerous enhancement to the language, one of them being *extension methods*. An extension method is a static method that defines an operation on and extends the functionality of an existing type without the need to formally extend the type you wish to enhance via normal inheritance. The new extension method can be used on the target type in the same way as an ordinary instance method.

The ArrayList has three extension methods: AsQueryable(), defined by the System.Linq.Queryable class, Cast<TResult>() and OfType<TResult>(), which are defined by the System.Linq.Enumerable class.

Defensive Coding Using the Façade Software Pattern

The non-generic ArrayList collection, found in the System.Collections namespace, can contain any type of object: objects in — objects out, but his flexibility comes at a price. If you store a mixture of object types in an ArrayList collection, you must, if you're thinking about accessing interface members other than those specified by the Object class, keep track of such types so as not to throw an exception in your code. It's often desirable when programming with non-generic, old-school collections, to create a type-safe collection that only allows the insertion of a specific type of object.

There are several approaches you can take to creating a custom list collection that enforces type-safety. First, you could extend the ArrayList class and override all the public members it exposes. Note that you would need to override all such members because a failure to do so would expose the base ArrayList members not overridden. A second approach would be to extend the System.Collections.CollectionBase class. This is actually the approach recommended by Microsoft when you need to create a strongly-typed collection, but I will postpone its discussion until I formally cover custom collections in Chapter 18: Creating Custom Collections.

A third approach, and one that's fairly easy to implement, is to create a façade or wrapper class that contains an ArrayList and provides implementations for only those methods you need. Figure 6-6 gives the UML class diagram for a custom collection named DogList which uses the façade software design pattern to encapsulate an ArrayList collection and provide a measure of type safety.

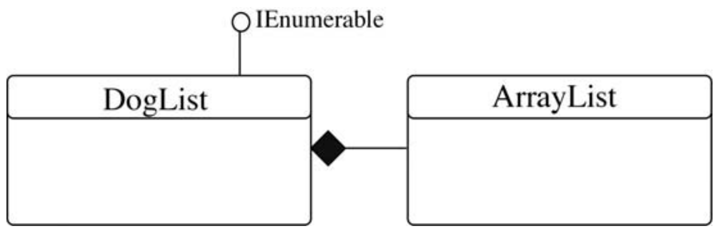


Figure 6-6: DogList Class Diagram

Referring to figure 6-6 — the DogList class contains an ArrayList by value and implements the IEnumerable interface. By implementing the IEnumerable interface, a DogList can be iterated via the `foreach` statement.

As its name implies, the DogList class will ensure that objects inserted into its ArrayList collection are of type Dog, the code for which is given in example 6.5.

6.5 Dog.cs

```
using System;

public class Dog : IComparable {
  private string _first_name;
  private string _last_name;
  private string _breed;

  public Dog(string breed, string f_name, string l_name){
    _breed = breed;
    _first_name = f_name;
    _last_name = l_name;
  }

  public string FirstName {
    get {  return _first_name; }
    set {  _first_name = value; }
  }

  public string LastName {
    get {  return _last_name; }
    set {  _last_name = value; }
  }

  public string Breed {
    get {  return _breed; }
    set {  _breed = value; }
  }

  public string FullName {
    get {  return FirstName + " " + LastName; }
  }
```



```
32
33     public string BreedAndFullName {
34       get {  return Breed + ": " + FullName; }
35     }
36
37     public override string ToString(){
38       return this.BreedAndFullName;
39     }
40
41     public int CompareTo(object obj){
42       if(obj is Dog){
43         return this.LastName.CompareTo(((Dog)obj).LastName);
44       }else{
45         throw new ArgumentException("Object being compared is not a Dog!");
46       }
47     }
48   }
```

Referring to example 6.5 — the Dog class implements the IComparable interface so that instances of Dog can be compared to each other. This is required because I want to implement the Sort() method in the DogList class. The IComparable interface specifies a method named CompareTo(object obj) which I have implemented beginning on line 41. The first thing the CompareTo() method does is to check the type of incoming object to ensure it's a Dog. If the comparison succeeds, the current instance, represented by the `this` pointer, is compared with the obj parameter — I am comparing LastName properties in this case. If the incoming argument is not of type Dog, an ArgumentException is thrown.

I have also overridden the Object.ToString() method, on line 37, which simply returns the value of the BreedAndFullName property.

Example 6.6 gives the code for the DogList class.

6.6 DogList.cs

```
using System;
using System.Collections;

public class DogList : IEnumerable {
   private ArrayList _list = null;


   public DogList(int size){
     _list = new ArrayList(size);
   }

   public DogList():this(25){   }

   public int Count {
     get {  return _list.Count; }
   }


   public Dog this[ int index] {
     get {
        return (Dog)_list[ index];
     }
     set {
       _list[ index]  = value;
     }

   }

   public int Add(Dog d){
     return _list.Add(d);
   }

   public void Remove(Dog d){
     _list.Remove(d);
   }

   public void RemoveAt(int index){
     _list.RemoveAt(index);
   }

   public void Reverse(){
     _list.Reverse();
   }

   public IEnumerator GetEnumerator(){
    return _list.GetEnumerator();
   }
```



```

   public void Sort(){
     _list.Sort();
   }
}
```

Referring to example 6.6 — the DogList class implements IEnumerable and encapsulates an ArrayList collection. I have provided implementations for just a small handful of the methods and properties found in the ArrayList class including the indexer, Add(), Remove(), RemoveAt(), Reverse(), and Sort(). Note how easy it is to implement the GetEnumerator() method with this particular approach. Example 6.7 shows the DogList class in action.

6.7 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        DogList list = new DogList();
6        list.Add(new Dog("Mutt", "Skippy", "Jones"));
7        list.Add(new Dog("French Poodle", "Bijou", "Jolie"));
8        list.Add(new Dog("Yellow Lab", "Schmoogle", "Miller"));
9        list.Add(new Dog("Mutt Lab", "Dippy", "Miller"));
10
11       for(int i = 0; i < list.Count; i++){
12         Console.WriteLine(list[ i]);
13       }
14
15       Console.WriteLine("-------------------------------");
16       list.Sort();
17
18       foreach(Dog d in list){
19         Console.WriteLine(d);
20       }
21
22       Console.WriteLine("-------------------------------");
23       list.Reverse();
24
25       foreach(Dog d in list){
26         Console.WriteLine(d);
27       }
28     }
29   }
```

Referring to example 6.7 — a DogList instance is created on line 5, followed by the insertion of four Dog objects into the collection. On line 11 I demonstrate the DogList indexer by iterating over the list with a `for` loop. On line 16 I make a call to the Sort() method which sorts the contents of the list by last name. Next, I iterate over the list with the help of the enumerator and the `foreach` statement. Finally, on line 23, I call Reverse() to reverse the list elements, and again print the list contents to the console with the `foreach` statement. Figure 6-7 shows the results of running this program.

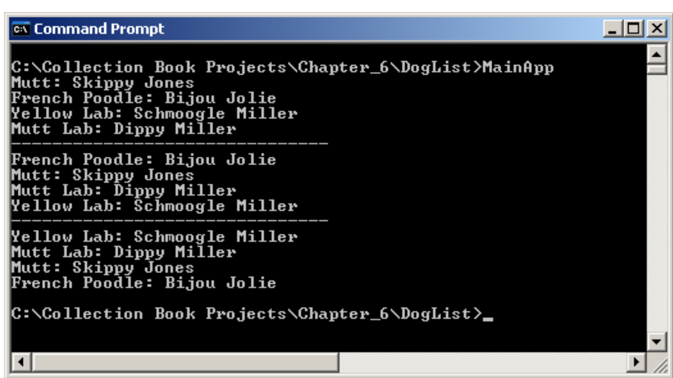


Figure 6-7: Results of Running Example 6.7

To be clear, the approaches described above would only apply if you were forced to use non-generic collections. This type of programming has been superseded by the introduction of generic collections.

Quick Review

The non-generic ArrayList collection can hold any type of object. It implements the IEnumerable, ICollection, IList, and ICloneable interfaces. The IEnumerable interface together with the IEnumerator interface enable the ArrayList elements to be iterated in a standardized, sequential fashion, beginning with the first element of the list and going forward. The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection by an index, just like an array. The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection.

You can employ one of three approaches to create a type-safe list collection: 1) extend the ArrayList class and override all its public members, 2) extend the CollectionBase class, which is the recommended approach, or 3) use the façade design pattern and create a wrapper class that encapsulates an ArrayList collection and provides implementations for the most often-used interface methods. Note that all these approaches are superseded by the introduction of generic collection classes.

The Generic List<T> Collection

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter T.

In this section I will discuss the differences between the ArrayList and List<T> inheritance hierarchies and highlight the benefits derived from utilizing the generic List<T> class.

List<T> Inheritance Hierarchy

Figure 6-8 gives the UML class diagram for the expanded inheritance hierarchy of the List<T> collection class.

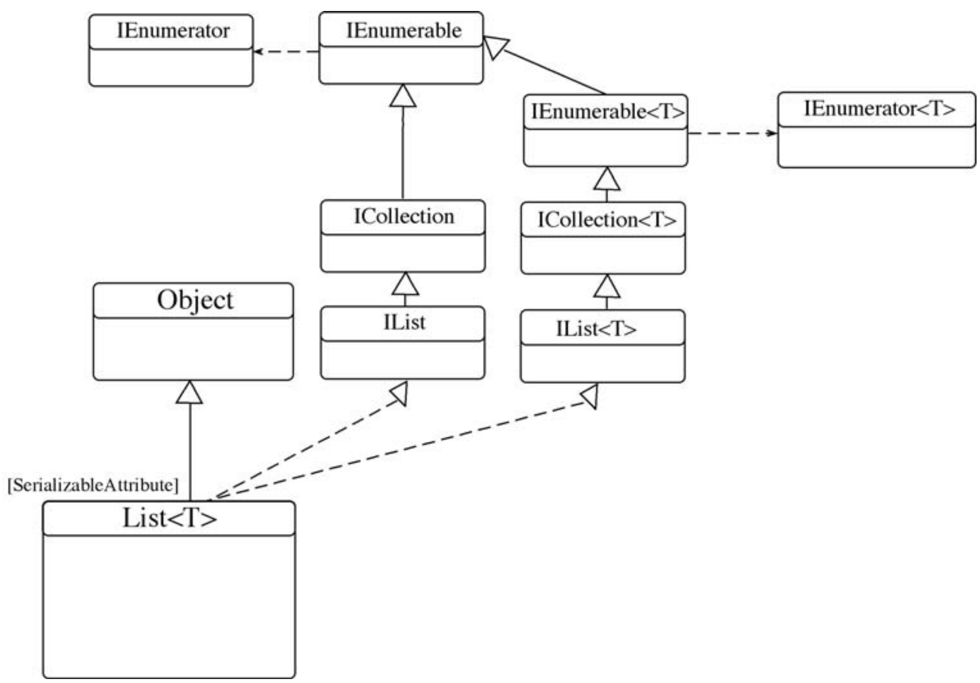


Figure 6-8: List<T> Inheritance Hierarchy

Referring to figure 6-8 — the List<T> class implements the same interfaces as the ArrayList class plus the following additional interfaces: IEnumerable<T>, ICollection<T>, and IList<T>. The IEnumerable<T> interface has a

dependency on the IEnumerator<T> interface. Notice also that IEnumerable<T> extends IEnumerable. The following sections discuss the functionality provided by these interfaces.

Functionality Provided by the IEnumerable<T> and IEnumerator<T> Interfaces

The IEnumerable<T> and IEnumerator<T> interfaces together expose the enumerator for collections of a specified type. The IEnumerable<T> interface declares two overloaded versions of the GetEnumerator() method: One returns an IEnumerator object and the other returns and IEnumerator<T> object.

In addition to these two methods, the IEnumerable<T> sports an additional 43 extension methods, most of which are defined by the System.Linq.Enumerable class. (Compare this with the three extension methods defined for the non-generic IEnumerable interface.) As you can see, IEnumerable<T> offers a lot more functionality than IEnumerable.

Functionality Provided by the ICollection<T> Interface

The ICollection<T> interface extends IEnumerable<T> and serves as the base interface for almost all of the generic collection classes. It supplies the Add(), Clear(), Contains(), CopyTo(), GetEnumerator(), and Remove() methods in addition to the Count and IsReadOnly properties.

Functionality Provided by the IList<T> Interface

The IList<T> interface extends ICollection<T> and represents a strongly typed collection of elements that can be accessed via an index. Individual element access is provided by an indexer. (**Note:** It is the implementation of the IList or IList<T> interfaces that enable a collection to be accessed with an indexer like an ordinary array.)

Benefits of Using List<T> vs. ArrayList

The List<T> class provides two primary benefits over the non-generic ArrayList class. First, you get a significant performance improvement when manipulating large collections of value-type object. Value-type objects normally require a boxing and unboxing operation when used in a non-generic collection. But when a value-type is specified for the type parameter <T> in a generic collection, an optimized version of the collection is generated based on the specified type. This customization of the generated generic type eliminates the need to box and unbox value-type objects in the collection.

The second primary benefit to using the List<T> class is the wide array of extension methods you can use to manipulate list elements. These extension methods are simply not available via the ArrayList class. Example 6.8 demonstrates the use of several extension methods on a list of integers.

6.8 ListExtensionMethodsDemo.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class ListExtensionMethodsDemo {

  private static void PrintList(List<int> list){
    for(int i=1; i<list.Count + 1; i++){
      if((i%10) > 0){
         Console.Write(list[i-1] + "\t");
      }else{
         Console.WriteLine(list[i-1] );
       }
     }
    Console.WriteLine();
  }

  public static void Main(){
    List<int> list = new List<int>();
    Random random = new Random();

    for(int i=0; i<50; i++){
      list.Add(random.Next(0, 1000));
    }

```



```
26        ListExtensionMethodsDemo.PrintList(list);
27        Console.WriteLine("----------------------------------");
28        Console.WriteLine("The sum of the list elements is: " + list.Sum());
29        Console.WriteLine("The average of the list elements is: " + list.Average());
30        Console.WriteLine("The maximum element value is: " + list.Max());
31        Console.WriteLine("The minimum element value is: " + list.Min());
32        Console.WriteLine("----------------------------------");
33        list.Sort();
34        ListExtensionMethodsDemo.PrintList(list);
35      }
36    }
```

Referring to example 6.8 — Note that for this example to work you must add the using directive on line 3 to access the System.Linq namespace. The Main() method begins on line 18. On line 19 I declare and instantiate a list of integers. On the following line I create a Random object and use it in the `for` loop beginning on line 22 to populate the list with integer values between 0 and 1000 inclusive. On line 26 a call to the static PrintList() method prints the elements in the list in a nice readable rectangular pattern.

On lines 28 through 31 I call the Sum(), Average(), Max(), and Min() extension methods respectively and print the results of each method call to the console. I then call the Sort() method to sort the list elements and print the results to the console. Figure 6-9 shows the results of running this program.

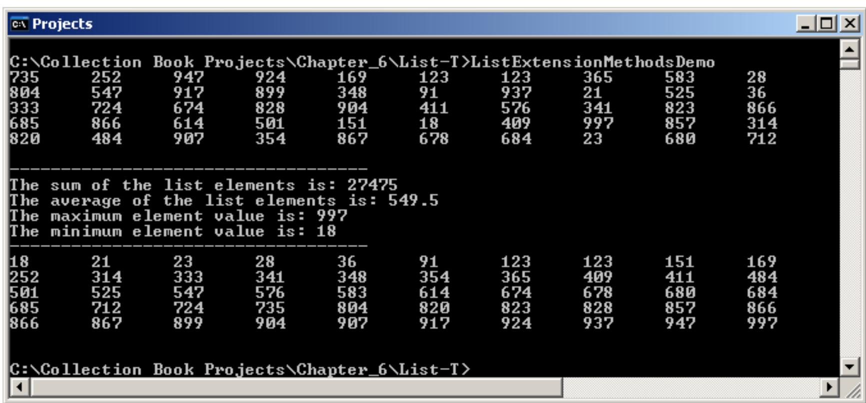


Figure 6-9: Results of Running Example 6.8

Quick Review

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter <T>.

The List<T> class offers several benefits over the non-generic ArrayList class: 1) faster performance when manipulating large collections of value-type objects, and 2) numerous extension methods defined by the System.Linq.Enumerable class.

Linked List Collections – How They Work

A linked list serves as the foundational data structure for the LinkedList<T> collection. In this section I will compare the performance characteristics of array-based lists to linked lists and show you how linked lists work under the covers.

Linked Lists vs. Array-Based Lists

Linked lists differ from array-based lists in many ways. An array-based list will always consume a certain amount of memory in the form of the array's initial capacity. In other words, when the list is created, its underlying

array is created to hold a specific number of elements. In many cases these array elements contain no data, at least not initially, but the space is reserved just the same. In the case of an array of reference types, the size of each element is equal to the virtual machine's memory word size (32-bits for example). For value types, the size of each element will equal the size of the data structure. Integers are 32 bits, longs are 64 bits, etc., and user-defined value types can be and usually are larger. Thus, a large array of user-defined value types might tie up a significant amount of memory. However, at least in the case of array-based collections, the use of memory is conserved by starting the array size small and growing it larger only upon demand. Conversely, a linked list creates element nodes only when needed, therefore conserving memory at the outset.

Another big difference between the two data structures is how elements are inserted and retrieved. In an array-based collection, element retrieval via an indexer is lightening quick and constant across the entire array. However, if an element needs to be inserted in the middle of the array, elements must be shifted to accommodate the insertion. This may take some time if the list contains many elements. On the other hand, element insertions in a linked list take a constant amount of time, but the time it takes to retrieve an element from a linked list depends on where in the list it's located and the number of elements the list contains.

Linked List Operation - The Circular Linked List

A linked list is a data structure based on linked nodes. Unlike an array, where elements are grouped together contiguously, a linked list's nodes may be anywhere in memory. The only way you can locate a particular element in a linked list is to start at the first element (head) and search forward, or start at the last element (tail) and search backwards, moving from node to node via the link each node maintains to the next, as figure 6-10 illustrates.

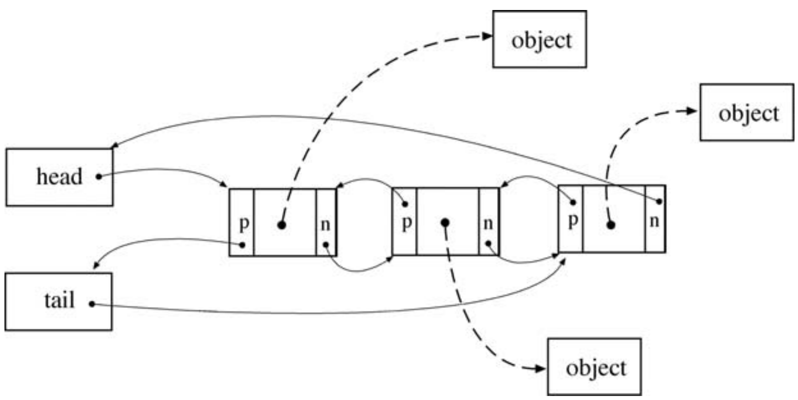


Figure 6-10: Circular, Doubly Linked List Data Structure

Referring to figure 6-10 — this type of linked list is referred to as a circular, doubly linked list. It's a doubly linked list because each node contains two references that maintain location information for the next node in the list and the previous node. It's circular because the first node in the list's previous reference is set to point to the tail, and the last node's next reference is set to point to the head. In this way one can "walk" the list from the first element to the last and return to the first element.

I'd like to illustrate these concepts better with some example code. Examples 6.9 through 6.11 demonstrate how a circular linked list might be constructed. Let's start by looking at the code for the node.

6.9 Node.cs

```
public class Node<T> {
  private Node<T> _previous;
  private Node<T> _next;
  private T _value;

  public Node<T> Previous {
    get { return _previous; }
    set { _previous = value; }
  }

  public Node<T> Next {
    get { return _next; }
    set { _next = value; }
  }

  public T Value {
```



```
      get {  return _value; }
      set {  _value = value; }
    }
  }
```

Referring to example 6.9 — the Node class contains three properties: Previous, Next, and Value, that correspond to the fields _previous, _next, and _value. Note how the fields _previous and _next are of type Node<T>. This is an example of when you are allowed to define a field to be the same type as the class you are defining.

Example 6.10 gives the code for the CircularLinkedList data structure.

6.10 CircularLinkedList.cs

```
using System;

public class CircularLinkedList<T> {
    // private fields
    private Node<T> _head = null;
    private Node<T> _tail = null;
    private int _count = 0;

    // constructor
    public CircularLinkedList() {}

    // read-only  properties
    public int Count {
        get {  return _count; }
    }

    public Node<T> First {
        get {  return _head; }
    }

    public Node<T> Last {
        get {  return _tail; }
    }

    // methods
    public Node<T> AddFirst(T value) {
        if (value == null) throw new ArgumentNullException();
        Node<T> node = new Node<T>();
        node.Value = value;
        if (_head == null) {
            // list is empty
            _head = node;
            _tail = node;
            node.Previous = _tail;
            node.Next = _head;
            _count++;
        }  else {
            _head.Previous = node;
            node.Next = _head;
            node.Previous = _tail;
            _tail.Next = node;
            _head = node;
            _count++;
        }
        return node;
    }

    public Node<T> AddLast(T value) {
        if (value == null) throw new ArgumentNullException();
        if (_tail == null) {
            // list is empty
            return this.AddFirst(value);
        }
        Node<T> node = new Node<T>();
        node.Value = value;
        node.Next = _tail.Next;
        node.Previous = _tail;
        _tail.Next = node;
        _tail = node;
        _count++;
        return node;
    }

    public Node<T> AddBefore(Node<T> node, T value) {
        if ((value == null) || (node == null)) throw new ArgumentNullException();
        if (node == _head) {
            return this.AddFirst(value);
        }

```

```
            Node<T> new_node = new Node<T>();
            new_node.Value = value;
            new_node.Previous = node.Previous;
            node.Previous = new_node;
            new_node.Next = node;
            new_node.Previous.Next = new_node;
            _count++;
            return new_node;
        }

        public Node<T> AddAfter(Node<T> node, T value) {
            if ((value == null) || (node == null)) throw new ArgumentNullException();
            if (node == _tail) {
                return this.AddLast(value);
            }

            Node<T> new_node = new Node<T>();
            new_node.Value = value;
            new_node.Previous = node;
            new_node.Next = node.Next;
            node.Next.Previous = new_node;
            node.Next = new_node;
            _count++;
            return new_node;
        }

        public void Remove(Node<T> node) {
            if (node == null) throw new ArgumentNullException();
         if(this.Find(node)){
            node.Next.Previous = node.Previous;
            node.Previous.Next = node.Next;
            if (node == _head) {
               _head = node.Next;
            }
            if (node == _tail) {
               _tail = node.Previous;
            }
            node.Next = null;
            node.Previous = null;
            _count--;
            if(_count == 0){
              _head = null;
              _tail = null;
            }
         } else {   // throw an exception because the node does not belong to this list anymore...
             throw new InvalidOperationException("Node does not belong to this linked list!");
         }
        }

        public Node<T> Find(T value) {
            if (value == null) throw new ArgumentNullException();
            Node<T> current_node = _head;
            for (int i = 0; i < _count; i++) {
                if (current_node.Value.Equals(value)) {
                    return current_node;
                } else {
                    current_node = current_node.Next;
                }
            }
            return null; // return null if value not found in list
        }

      private bool Find(Node<T> node){
         if(node == null) throw new ArgumentNullException();
         if(_count == 0) return false;
         if(_head == node) return true;
         Node<T> temp_node = _head;
         for(int i = 0; i < _count; i++){
           if(temp_node.Next == node) return true;
          temp_node = temp_node.Next;
         }
         return false;
      }

   } // end CircularLinkedList<T> class definition
```

Referring to example 6.10 — the CircularLinkedList<T> class defines three private fields: _head, _tail, and _count, and three public properties: First, which corresponds to the _head field; Last, which corresponds to the _tail field, and Count, which of course corresponds to the _count field. I've also included six public methods: AddFirst(), AddLast(), AddBefore(), AddAfter(), Remove(), and an overloaded Find(). (The property and method names I've

used here match the property and method names of the System.Collections.Generic.LinkedList<T> class, which I discuss later in the chapter.)

Again referring to example 6.10 — let’s take a closer look at the AddFirst() method. As its name implies, the AddFirst() method inserts the value as the first element in the list. The first thing I do is check the validity of the incoming value argument by making sure it’s not null. If it is null, I throw an ArgumentNullException. Past that hurdle, I create a new Node<T> object and assign its Value property to equal the incoming value argument. Next, I check to see if the _head field is null. If it is, the list is currently empty and I make the insertion based on that assumption, else, the list contains at least one element, and the insertion must take into account the presence of an existing node.

When you do programming like this, you’ll find yourself drawing diagrams on scrap pieces of paper or in your engineering notebook to ensure you’ve accounted for all the references that must be set on each affected node, including the _head and _tail fields. Figures 6-11 and 6-12 show several pages from my engineering notebook when I wrote this code.

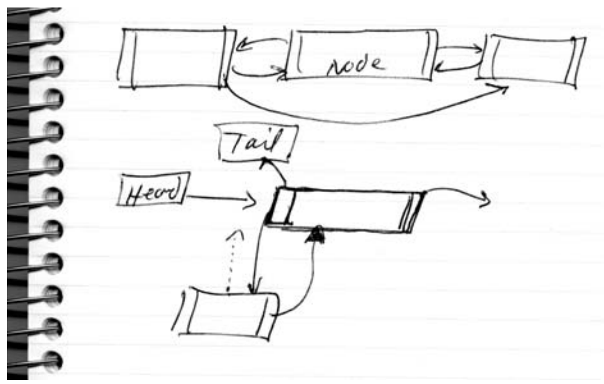


Figure 6-11: Linked List Insertion Notes

(Looking at those notes reminds me of learning to count and do math with the help of one’s fingers!)

The AddLast() method works similar to the AddFirst() method, only it inserts the value as the last element in the list. The AddBefore(Node<T> node, T value) and AddAfter(Node<T> node, T value) methods take an existing node as their first argument and the value to be inserted as the second argument. These methods are used in conjunction with the Find() method, where the list must be searched for a particular value, and then the incoming value inserted into the list either before or after.

The Remove() method simply removes the indicated node from the list and subtracts one from the list count.

Example 6.11 shows the CircularLinkedList class in action.

6.11 MainApp.cs (Demonstrating CircularLinkedList)

```
using System;

public class MainApp {
  private static CircularLinkedList<int> list = new CircularLinkedList<int>();
  private static Node<int> current_node;

  // utility method
    private static void PrintListValues(){
      current_node = list.First;
      for(int i = 0; i < list.Count; i++){
        Console.Write(current_node.Value + " ");
        current_node = current_node.Next;
      }
      Console.WriteLine();
    }

  public static void Main(){

    // Test AddFirst() method
    MainApp.list.AddFirst(3);
    MainApp.list.AddFirst(2);
    MainApp.list.AddFirst(1);
    MainApp.PrintListValues();

    // Test Remove() method
    MainApp.list.Remove(current_node.Previous);
    MainApp.PrintListValues();

```

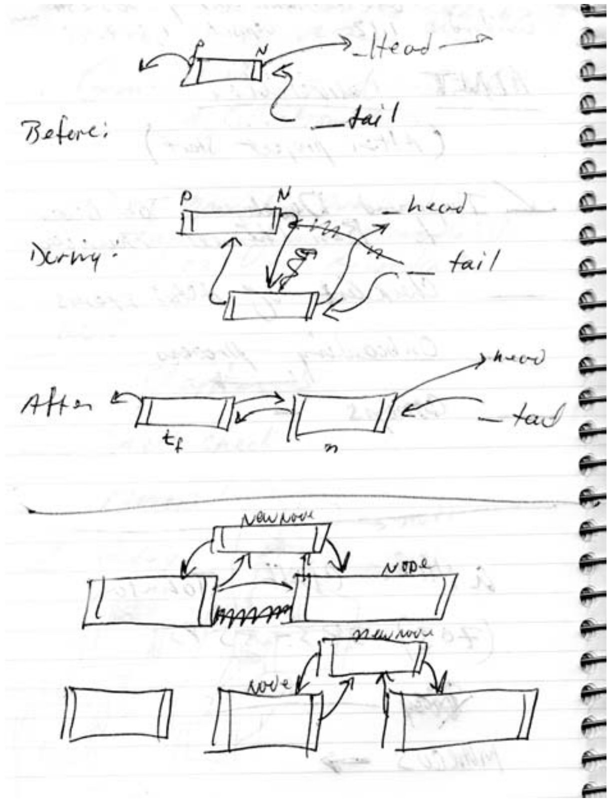



Figure 6-12: Linked List Insertion Notes

```
    // Test AddBefore() method
    MainApp.list.AddBefore(current_node.Previous, 4);
    MainApp.PrintListValues();

    // Test AddAfter() method
    MainApp.list.AddAfter(MainApp.list.Find(4), 5);
    MainApp.PrintListValues();

    // Remove last element
    MainApp.list.Remove(MainApp.list.Last);
    MainApp.PrintListValues();

    // Remove first element
    MainApp.list.Remove(MainApp.list.First);
    MainApp.PrintListValues();

    // Test AddLast() method
    MainApp.list.AddLast(6);
    MainApp.PrintListValues();
    Console.WriteLine("List has " + MainApp.list.Count + " items");

    // Test AddAfter() method again
    MainApp.list.AddAfter(MainApp.list.Find(6), 7);
    MainApp.PrintListValues();

    // Test AddBefore method again
    MainApp.list.AddBefore(MainApp.list.Find(4), 3);
    MainApp.PrintListValues();

    // Let's step forward through the list
    Console.Write("First element = " + MainApp.list.First.Value + " ");
    Console.Write("Second element = " + MainApp.list.First.Next.Value + " ");
    Console.Write("Third element = " + MainApp.list.First.Next.Next.Value + " ");
    Console.Write("Forth element = " + MainApp.list.First.Next.Next.Next.Value + " ");
    Console.Write("Fifth element = " + MainApp.list.First.Next.Next.Next.Next.Value + " ");
    Console.Write("Next element = " + MainApp.list.First.Next.Next.Next.Next.Next.Value + " ");
```



```
65       Console.WriteLine();
66
67       // Now backwards
68       Console.Write("Last element = " + MainApp.list.Last.Value + " ");
69       Console.Write("Next element = " + MainApp.list.Last.Previous.Value + " ");
70       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Value + " ");
71       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Value + " ");
72       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Previous.Value + " ");
73
74      Console.WriteLine("\n--------------------------------------------");
75      Console.WriteLine("Number of elements in the list: " + MainApp.list.Count);
76      while(MainApp.list.Count > 0){
77         MainApp.list.Remove(MainApp.list.Last);
78      }
79      Console.WriteLine("Number of elements in the list: " + MainApp.list.Count);
80     }
81   }
```

Referring to example 6.11 — this MainApp class is structured slightly differently than previous MainApp examples. It contains two private fields and a method named PrintListValues(). The PrintListValues() method simply walks the list and prints each value to the console.

The Main() method begins on line 17. The first thing I do is exercise the AddFirst() method by inserting three integers into the list followed by a call to the PrintListValues() method. I follow this with a test of the various other methods defined by the CircularLinkedList class.

On lines 58 through 72 I demonstrate how to walk the list first going forward, via the Next references, then backwards using the Previous references.

Figure 6-13 shows the results of running example 6.11 demonstrating the use of the CircularLinkedList<T> class.

Figure 6-13: Results of Running Example 6.11 Testing the Circular Linked List

Quick Review

A linked list stores its elements in non-contiguous nodes which are linked together via Next and Previous references. The individual list nodes might be located anywhere in memory. To locate a specific node in a linked list, you must either start at the Head or First node and traverse the list forward, or start at the Tail or Last node and traverse the list backwards. A linked list conserves memory by storing data in individual nodes and grows the list one node at a time when needed.

The Generic LinkedList<T> Collection

Now that you understand how a linked list works, you'll understand how the generic LinkedList<T> class operates. As its name implies, the LinkedList<T> class stores elements in a linked list structure. Elements can be accessed sequentially beginning at the First element and walking the list forward, or starting at the Last element and walking the list backward.

The LinkedList<T> class provides a whole lot more functionality than the CircularLinkedList example presented in the previous section. Like most of the collections in the System.Collections.Generic namespace, the LinkedList<T> class can be manipulated with extension methods defined by the System.Linq.Enumerable class.

LinkedList<T> is Non-Circular!

An important feature to understand up front about the LinkedList<T> class is that it's not a circular list. In other words, the Next reference of the Last node points to null, and the Previous reference of the First node points to null as well.

The LinkedList<T> Inheritance Hierarchy

The LinkedList<T> class's inheritance hierarchy is shown in figure 6-14.

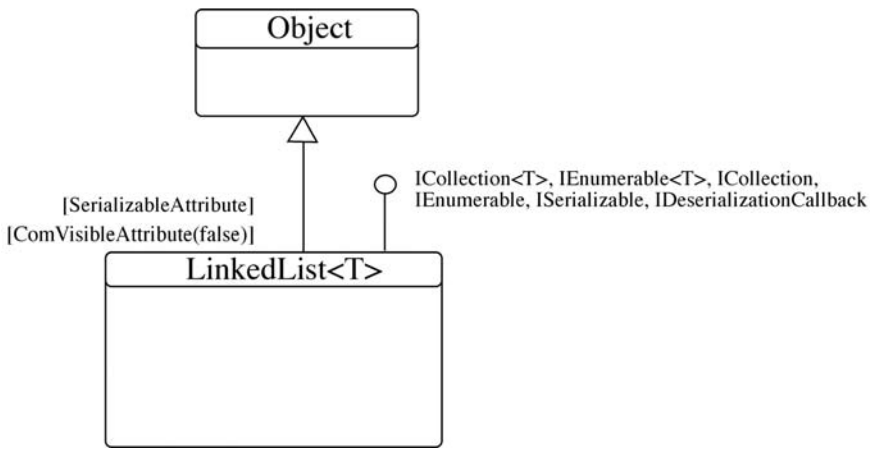


Figure 6-14: LinkedList<T> Inheritance Hierarchy

Referring to figure 6-14 — the LinkedList<T> class implements the IEnumerable, IEnumerable<T>, ICollection, ICollection<T>, ISerializable, and IDeserializationCallback interfaces.

Functionality Provided By The IEnumerable and IEnumerable<T> Interfaces

The IEnumerable and IEnumerable<T> interfaces, along with their related IEnumerator and IEnumerator<T> interfaces expose the enumerator for the linked list structure. You can walk the list from the first to the last element on the list with the `foreach` statement. You can also access the enumerator directly by calling the GetEnumerator() method.

The thing to remember when using an enumerator with any collection class is that any attempt to modify the underlying collection while traversing the collection with the help of its enumerator will invalidate the enumerator and its behavior from that point forward cannot be predicted. If you need to control access to a collection while traversing its contents you must synchronize access to the collection object. This topic is discussed in detail in Chapter 14: Collections and Threads.

Functionality Provided by the ICollection and ICollection<T> Interfaces

The ICollection interface extends IEnumerable; the ICollection<T> interface extends IEnumerable<T>. Together the ICollection and ICollection<T> interfaces allow the LinkedList<T> class to be treated as a collection object, publishing the Add(), Clear(), Contains(), CopyTo(), GetEnumerator(), and Remove() methods.

Functionality Provided by the ISerializable and IDeserializationCallback Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the programmers of the LinkedList<T> class had to implement custom serialization routines over and above what the SerializableAttribute provided. I discuss custom serialization and deserialization in detail in Chapter 17: Collections and I/O.

LinkedList<T> Collection in Action

Example 6.12 demonstrates the use of the LinkedList<T> collection class. This example closely resembles the code used to demonstrate the CircularLinkedList in the previous section, however, I've modified it to work correctly with the non-circular LinkedList<T> class.

6.12 MainApp.cs (LinkedList<T> Demo)

```
using System;
using System.Collections.Generic;

public class MainApp {
  private static LinkedList<int> list = new LinkedList<int>();
  private static LinkedListNode<int> current_node;

  // utility method
    private static void PrintListValues(){
      current_node = list.First;
      for(int i = 0; i < list.Count; i++){
        Console.Write(current_node.Value + " ");
        current_node = current_node.Next;
      }
      Console.WriteLine();
    }

  public static void Main(){

    // Test AddFirst() method
    MainApp.list.AddFirst(3);
    MainApp.list.AddFirst(2);
    MainApp.list.AddFirst(1);
    MainApp.PrintListValues();

    // Test Remove() method
    MainApp.list.Remove(MainApp.list.Last);
    MainApp.PrintListValues();

    // Test AddBefore() method
    MainApp.list.AddBefore(MainApp.list.Last, 4);
    MainApp.PrintListValues();

    // Test AddAfter() method
    MainApp.list.AddAfter(MainApp.list.Find(4), 5);
    MainApp.PrintListValues();

    // Remove last element
    MainApp.list.Remove(MainApp.list.Last);
    MainApp.PrintListValues();

    // Remove first element
    MainApp.list.Remove(MainApp.list.First);
    MainApp.PrintListValues();

    // Test AddLast() method
    MainApp.list.AddLast(6);
    MainApp.PrintListValues();
    Console.WriteLine("List has " + MainApp.list.Count + " items");

    // Test AddAfter() method again
    MainApp.list.AddAfter(MainApp.list.Find(6), 7);
    MainApp.PrintListValues();

    // Test AddBefore method again
    MainApp.list.AddBefore(MainApp.list.Find(4), 3);
    MainApp.PrintListValues();

    // Let's step forward through the list
    Console.Write("First element = " + MainApp.list.First.Value + " ");
    Console.Write("Second element = " + MainApp.list.First.Next.Value + " ");
    Console.Write("Third element = " + MainApp.list.First.Next.Next.Value + " ");
    Console.Write("Forth element = " + MainApp.list.First.Next.Next.Next.Value + " ");
```



```
64        Console.Write("Fifth element = " + MainApp.list.First.Next.Next.Next.Next.Value + " ");
65        Console.WriteLine();
66
67        // Now backwards
68        Console.Write("Last element = " + MainApp.list.Last.Value + " ");
69        Console.Write("Next element = " + MainApp.list.Last.Previous.Value + " ");
70        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Value + " ");
71        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Value + " ");
72        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Previous.Value + " ");
73     }
74   }
```

Referring to example 6.12 — a LinkedList<int> reference is declared and initialized on line 5. On line 6, an instance of LinkedListNode<int> is declared for use later in the program. In the body of the Main() method, I put the linked list through its paces. Figure 6-15 shows the results of running this program.

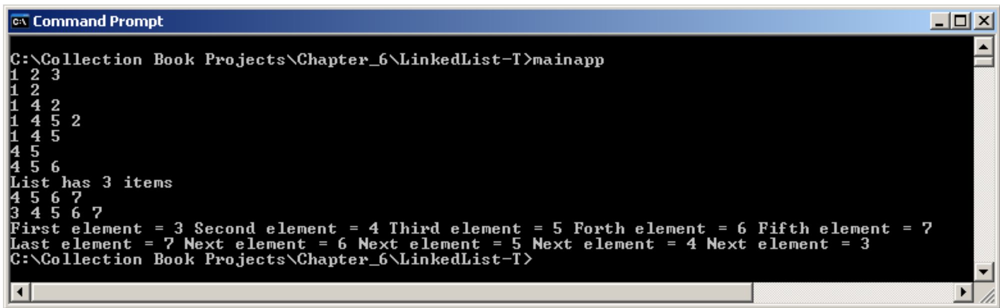


Figure 6-15: Results of Running Example 6.12

Quick Review

The generic LinkedList<T> class implements a non-circular, doubly-linked list collection. Elements of the LinkedList<T> class are stored as individual nodes of type LinkedListNode<T>. You may traverse the linked list structure manually, starting at the First node and moving forward, or at the Last node and moving backwards through the list, moving from node to node via the Next or Previous references as required. You can also traverse the list nodes automatically via its enumerator.

Summary

Array-based lists, as their name implies, feature arrays as their fundamental data structure. Array-based lists are created with an initial capacity and can grow in size automatically to accommodate additional elements.

The non-generic ArrayList collection can hold any type of object. It implements the IEnumerable, ICollection, IList, and ICloneable interfaces. The IEnumerable interface together with the IEnumerator interface enable the ArrayList elements to be iterated in a standardized, sequential fashion, beginning with the first element of the list and going forward. The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection via an index, just like an array. The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection.

You can employ one of three approaches to create a type-safe list collection: 1) extend the ArrayList class and override all its public members, 2) extend the CollectionBase class, which is the recommended approach, or 3) use the façade design pattern and create a wrapper class that encapsulates an ArrayList collection and provides implementations for the most often-used interface methods. Note that all these approaches are superseded by the introduction of generic collection classes.

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter <T>.

The List<T> class offers several benefits over the non-generic ArrayList class: 1) faster performance when manipulating large collections of value-type objects, and 2) numerous extension methods defined by the System.Linq.Enumerable class.

A linked list stores its elements in non-contiguous nodes which are linked together via Next and Previous references. The individual list nodes might be located anywhere in memory. To locate a specific node in a linked list, you must either start at the Head or First node and traverse the list forward, or start at the Tail or Last node and traverse the list backwards. A linked list conserves memory by storing data in individual nodes and grows the list one node at a time when needed.

The generic LinkedList<T> class implements a doubly-linked, non-circular linked list collection. Elements of the LinkedList<T> class are stored as individual nodes of type LinkedListNode<T>. You may traverse the linked list structure manually, starting at the First node and moving forward, or at the Last node and moving backwards through the list, moving from node to node via the Next or Previous references as required. You can also traverse the list nodes automatically via its enumerator.

References

Ryan Stephens, et. al., *C++ Cookbook*, O'Reilly Media, Inc., Sebastopol, CA. ISBN-13: 978-0-596-00761-4

Erich Gamma, et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-63361-2

Don Box. *Essential COM*, Addison-Wesley, Boston, Massachusetts. ISBN: 0-201-63446-5

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

Chapter 7

Contax T

Acrobat Man

Stacks

Learning Objectives

- *Describe the operation of a stack*
- *Describe the characteristics of Last In/First Out (LIFO) processing*
- *List at least four examples of applications that require stacks*
- *State the type of data structure used to implement the Stack and Stack<T> classes*
- *Describe what it means to push items onto a stack*
- *Describe what it means to pop items off of a stack*
- *Describe the behavior of a pop operation*
- *List and describe the members of the Stack and Stack<T> classes*
- *Use the non-generic Stack class in a program*
- *Use the generic Stack<T> class in a program*
- *Describe the functionality provided by each interface implemented by the Stack class*
- *Describe the functionality provided by each interface implemented by the Stack<T> class*

Introduction

Stacks play a critical role in the world of computers. Microprocessors and virtual machines utilize stacks (stack frames) to implement procedure call chaining; compilers utilizes stacks to parse programming languages; application software might use a stack to implement an operation undo capability.

In this chapter I will explain how stacks work and show you an example of a custom coded stack so you can see how they work internally. I'll then discuss the Stack and Stack<T> classes in detail and present a comprehensive, non-trivial example showing each of these classes in action.

When you finish this chapter you'll have a solid understanding of how stacks work and why they are an important data structure.

Stack Operations

A stack is a special kind of list whose elements or items are stored and accessed in last-in/first-out (LIFO) sequence. All insertions and deletions to a stack occur at only one end of the list. The business end of a stack goes by a special name: "Top".

Characteristic Stack Operations

A stack data structure supports two primary operations: *push* and *pop*. A third operation called *peek* also comes in handy. These operations are discussed in detail below.

Push

The push operation adds an item to the top of a stack. Subsequent calls to push add newer items to the top of the stack. The number of items contained in the stack increments by one with each push.

Pop

The pop operation removes the top element from the stack. The last item pushed onto the stack will be the first item popped off of the stack. (LIFO). The number of items contained in the stack is reduced by one with each pop.

Peek

The peek operation is used to examine the item at the top of the stack in place without removing the item.

An Illustration Will Help

Figure 7-1 shows a representation of a stack and the effects of several push and pop operations.

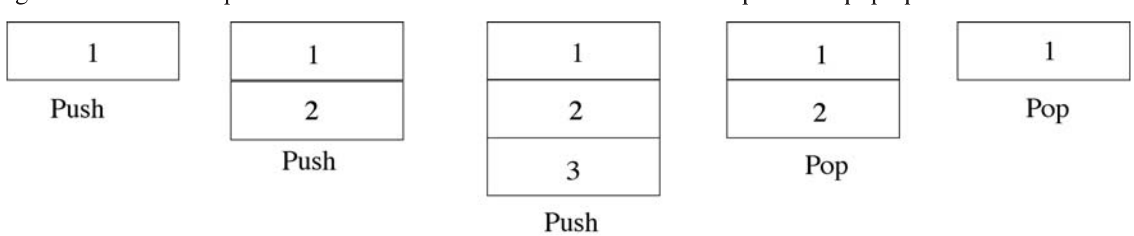


Figure 7-1: Stack Showing Effects of Push and Pop Operations

Referring to figure 7-1 — in this picture I have represented the stack as growing downward with each successive push operation. That is, the *top* of the stack grows downward with each successive push operation. After the first push operation item 1 sits on the top of the stack. After the second push operation item 2 sits on the top of the stack, and finally, after the third push operation, item 3 sits on top of the stack. Item 3 is the first element removed as a result of the first pop operation. Item 2 is removed as a result of the second call to pop, which leaves item 1 at the top of the stack.

What's Actually Being Pushed and Popped?

That's a good question, and since this is a book about C# and the .NET Framework, the answer is one of two things: 1) you're either going to push a value type object, or 2) a reference type object. It's important to know the difference between the two.

Pushing a Value Type Object onto a Stack

A value type object implements value type copy semantics. By this I mean that when one value type is assigned to another, the value of one is copied to the other. In the .NET Framework, value type objects are structures and are defined using the `struct` keyword. If the structure is complex and contains many fields, each field's value will be copied from one instance of the value type object to another. This value type copy behavior is implemented automatically by the .NET runtime environment.

Now, when you use the non-generic version of Stack, found in the System.Collections namespace, you will encounter a performance penalty when pushing value types onto the stack. This performance penalty occurs because value types must be boxed into objects before being pushed onto the non-generic stack. The end result is that a reference to the boxed value type is actually pushed onto the stack and the boxed value type object is created on the heap.

If, on the other hand, you use the generic Stack<T> class and specify a value type for 'T', the resulting data structure is optimized for that value type and no boxing or unboxing occurs. However, the performance penalty you incur with pushing and popping will be commensurate with the complexity of the value type in question.

Pushing a Reference Type Object onto a Stack

The result of pushing a reference type onto a stack is that the stack contains only references to objects in the heap. What you must be aware of in this situation is the number of active references that point to the same object. For example, suppose you have a reference R to object O. If you push R onto the stack, the top of the stack now points to O as well. Two references to O are now active. The danger of having too many active references to one object is that as long as there is one active reference to an object the .NET runtime garbage collector cannot free up and reclaim the memory for future use. This advice applies not only to the use of stacks, but to .NET programming in general.

Value Type Boxing in Action

Example 7.1 gives the code for a short program that pushes 25 million integers onto two different types of stacks: the non-generic System.Collections.Stack, and the generic System.Collections.Generic.Stack<T>,

7.1 PushValueTypeDemo.cs

```
using System;
using System.Collections;
using System.Collections.Generic;

public class PushValueTypeDemo {
  public static void Main(){
    Stack stack1 = new Stack();
    Stack<int> stack2 = new Stack<int>();
    const int COUNT = 25000000;

    DateTime start = DateTime.Now;
    for(int i = 0; i < COUNT; i++){
      stack1.Push(i);
    }
    TimeSpan elapsed_time = (DateTime.Now - start);
    Console.WriteLine("Time to push {0:N} integers to non-generic stack: {1}", COUNT, elapsed_time);

    start = DateTime.Now; // reset start time
```



```
19          for(int i = 0; i < COUNT; i++){
20            stack2.Push(i);
21          }
22          elapsed_time = (DateTime.Now - start);
23          Console.WriteLine("Time to push {0:N} integers to generic stack of integers: {1}",
24                            COUNT, elapsed_time);
25        }
26      }
```

Referring to example 7.1 — on lines 7 and 8 an instance each of Stack and Stack<int> is created followed by the definition of a constant named COUNT which is initialized to 25,000,000. On line 11, the DateTime.Now property is used to initialize the variable named `start`. In the first `for` loop which beings on the next line, 25 million integers are pushed onto the non-generic stack. When the `for` loop exits, the elapsed time is calculated and the results printed to the console. The same process is then repeated with the generic Stack<int>. Figure 7-2 shows the results of running this program.

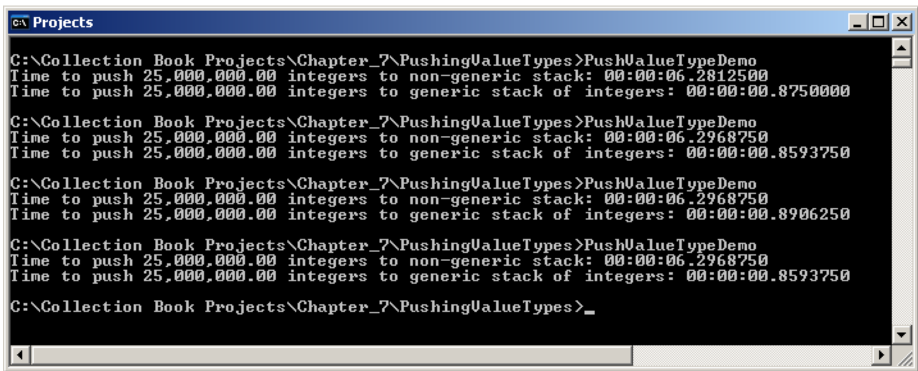


Figure 7-2: Results of Running Example 7.1

Referring to figure 7-2 — I executed the program four times. In each run the boxing of integer value types as they are pushed onto the non-generic stack extracted a performance penalty.

Disassembling Example 7.1

If you use the MSIL disassembler to disassemble the executable file created by compiling example 7.1 you'll get an output that looks similar to example 7.2. (Note: This is the disassembled Main() method.)

7.2 Disassembled Main() Method from Example 7.1

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       177 (0xb1)
  .maxstack  3
  .locals init (class [mscorlib]System.Collections.Stack V_0,
           class [System]System.Collections.Generic.Stack`1<int32> V_1,
           valuetype [mscorlib]System.DateTime V_2,
           int32 V_3,
           valuetype [mscorlib]System.TimeSpan V_4,
           bool V_5)
  IL_0000:  nop
  IL_0001:  newobj     instance void [mscorlib]System.Collections.Stack::.ctor()
  IL_0006:  stloc.0
  IL_0007:  newobj     instance void class [System]System.Collections.Generic.Stack`1<int32>::.ctor()
  IL_000c:  stloc.1
  IL_000d:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
  IL_0012:  stloc.2
  IL_0013:  ldc.i4.0
  IL_0014:  stloc.3
  IL_0015:  br.s       IL_002a
  IL_0017:  nop
  IL_0018:  ldloc.0
  IL_0019:  ldloc.3
  IL_001a:  box        [mscorlib]System.Int32
  IL_001f:  callvirt   instance void [mscorlib]System.Collections.Stack::Push(object)
  IL_0024:  nop
  IL_0025:  nop
  IL_0026:  ldloc.3
```



```
30        IL_0027:  ldc.i4.1
31        IL_0028:  add
32        IL_0029:  stloc.3
33        IL_002a:  ldloc.3
34        IL_002b:  ldc.i4     0x17d7840
35        IL_0030:  clt
36        IL_0032:  stloc.s    V_5
37        IL_0034:  ldloc.s    V_5
38        IL_0036:  brtrue.s   IL_0017
39        IL_0038:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
40        IL_003d:  ldloc.2
41        IL_003e:  call       valuetype [mscorlib]System.TimeSpan
[mscorlib]System.DateTime::op_Subtraction(valuetype [mscorlib]System.DateTime,valuetype
[mscorlib]System.DateTime)
42        IL_0043:  stloc.s    V_4
43        IL_0045:  ldstr      "Time to push {0:N} integers to non-generic stack: "
44        + "{1}"
45        IL_004a:  ldc.i4     0x17d7840
46        IL_004f:  box        [mscorlib]System.Int32
47        IL_0054:  ldloc.s    V_4
48        IL_0056:  box        [mscorlib]System.TimeSpan
49        IL_005b:  call       void [mscorlib]System.Console::WriteLine(string,
50                                                                  object,
51                                                                  object)
52        IL_0060:  nop
53        IL_0061:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
54        IL_0066:  stloc.2
55        IL_0067:  ldc.i4.0
56        IL_0068:  stloc.3
57        IL_0069:  br.s       IL_0079
58        IL_006b:  nop
59        IL_006c:  ldloc.1
60        IL_006d:  ldloc.3
61        IL_006e:  callvirt   instance void class [System]System.Collections.Generic.Stack`1<int32>::Push(!0)
62        IL_0073:  nop
63        IL_0074:  nop
64        IL_0075:  ldloc.3
65        IL_0076:  ldc.i4.1
66        IL_0077:  add
67        IL_0078:  stloc.3
68        IL_0079:  ldloc.3
69        IL_007a:  ldc.i4     0x17d7840
70        IL_007f:  clt
71        IL_0081:  stloc.s    V_5
72        IL_0083:  ldloc.s    V_5
73        IL_0085:  brtrue.s   IL_006b
74        IL_0087:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
75        IL_008c:  ldloc.2
76        IL_008d:  call       valuetype [mscorlib]System.TimeSpan
[mscorlib]System.DateTime::op_Subtraction(valuetype [mscorlib]System.DateTime,valuetype
[mscorlib]System.DateTime)
77        IL_0092:  stloc.s    V_4
78        IL_0094:  ldstr      "Time to push {0:N} integers to generic stack of in"
79        + "tegers: {1}"
80        IL_0099:  ldc.i4     0x17d7840
81        IL_009e:  box        [mscorlib]System.Int32
82        IL_00a3:  ldloc.s    V_4
83        IL_00a5:  box        [mscorlib]System.TimeSpan
84        IL_00aa:  call       void [mscorlib]System.Console::WriteLine(string,
85                                                                  object,
86                                                                  object)
87        IL_00af:  nop
88        IL_00b0:  ret
89      } // end of method PushValueTypeDemo::Main
```

Referring to example 7.2 — OK, before your eyes roll up into your skull take a deep breath. This will be easier to understand than you first think. It can be intimidating to decipher MSIL instructions your first time through. In this example, however, you'll only need to understand a handful of instructions. So here goes.

First, a word about the layout of the file. There are three columns. The leftmost column contains the IL address and other directives. The second column contains symbolic instructions, and the third column, if it contains anything, will have variable names, constant values, method names, object names, etc. These will be easy to figure out as you begin to get familiar with the code. I'm just going to discuss the first half of the code, the part that contains the first `for` loop of example 7.1.

Starting at the top of the listing on line 1, the text there signifies that this is a Main method. Line 2 contains an opening brace, and line 3 contains a directive that says this is the entry point. (.entrypoint). Line 4 contains a comment indicating the size of the code. The .maxstack directive on line 5 indicates the maximum amount of evaluation

stack space the program will utilize. You'll see the evaluation stack in action shortly. Lines 6 through 11 contain local variable declarations named V_0 through V_5. V_0 is the reference to the non-generic Stack object. V_1 is the reference to the generic Stack<int> object. (Here denoted as Stack<int32>.) V_2 is a DateTime variable and corresponds to the `start` variable declared in example 7.1. V_3 corresponds to the counting variable `i` declared in each of the for loops. V_4 corresponds to the TimeSpan variable `elapsed_time`. Finally, V_5 is the boolean value that is required to evaluate each of the `for` loops.

Line 12 contains a `nop` (no operation) instruction. Line 13 creates an instance of the non-generic Stack and leaves its reference on the evaluation stack. The `stloc.0` instruction on line 14 pops the value from the evaluation stack and loads it into local variable 0 (V_0). On line 15 an instance of the generic Stack<int> object is created and its reference is left on the evaluation stack. The `stloc.1` instruction on the next line pops the reference off the evaluation stack and assigns it to local variable 1 (V_1). On line 17, the `call` instruction makes a method call to the DateTime.get_Now() method. (*Under the covers, properties translate into method calls.*) The resulting value obtained from that call is pushed onto the evaluation stack, and the next instruction, stloc.2, pops the value off the evaluation stack and assigns it to local variable 2 (V_2). On line 19, the ldc.i4.0 instruction loads the value 0 onto the evaluation stack. The next instruction pops this value off the evaluation stack and assigns it to local variable 3 (V_3).

Now we're ready to get going on the loop. Line 21 contains a `br.s` instruction. This says to branch unconditionally to address IL_002a, which you'll find on line 33. The ldloc.3 instruction pushes the value of local variable 3 onto the evaluation stack. Next, the ldc.i4 instruction pushes the value of 0x17d7840 (hexadecimal for 25 million) onto the stack. This is followed by the ctl instruction (compare less than). So, the first time around 0 is less than 25 million, so the result will be true or 1 and this value is pushed onto the evaluation stack. On line 36, the stloc.s instruction pops this value from the stack and assigns it to local variable V_5 (the boolean variable). This value is then pushed back onto the stack in preparation for the next instruction on line 38 which is brture.s which tells the VM to branch to address IL_0017 if the value on the top of the evaluation stack is 1. Going to line 22 we see a nop instruction. On line 23 the ldloc.o loads the value of local variable 0 (V_0 -- the reference to the non-generic stack) onto the evaluation stack. Next, the ldloc.3 instruction loads the value of local variable 3 (the counting variable i, which is zero now.) onto the evaluation stack. On line 25, the box instruction boxes the value on top of the evaluation stack and then pushes it onto the stack on the next line with a callvirt instruction to the non-generic Stack.Push() method. Following two nop instructions the value of local variable 3 is pushed onto the evaluation stack followed by the instruction on line 30, ldc.i4.1, which pushes the value 1 onto the evaluation stack. These two values are added with the add instruction on line 31 and the result is popped from the stack and assigned to local variable 3 (V_3). In this way the counting variable i is incremented by one. Thus, the loop repeats in this fashion for 25 million iterations.

When, after 25 million iterations, the result of the comparison of the counting variable i and the constant COUNT results in false, the brtrue.s instruction will fail and execution will fall through to the instruction on line 39. This is where the program calculates how long it took to execute the first for loop. On line 39, a call to DateTime.get_Now() pushes the resulting value onto the evaluation stack. Next, on line 40, the value of local variable V_2 is pushed onto the stack. (V_2 contains the `start` value.) The instruction on line 42 performs a TimeSpan subtraction using the two DateTime values on the stack leaving the result on top of the stack. The stloc.s instruction on line 43 pops this value off the stack and stores it in local variable V_4, which corresponds to the `elapsed_time` variable in example 7.1. On line 44, the ldstr (load string) instruction loads a reference to the string literal indicated in quotes on the evaluation stack. This is followed by the ldc.i4 instruction which load the value 0x17d7840 (25 million decimal) onto the evaluation stack followed by a call to the box instruction to box the value. Next, local variable V_4 is loaded onto the evaluation stack, and since a DateTime value is a value type, it's boxed as well. The state of the stack now is a reference to a string, a reference to a boxed integer, and a reference to a boxed DateTime value. Finally, on line 50, a call to the Console.WriteLine() method prints the string and the two values to the console.

Any questions? Note that this was a great exercise because you got to see not only how C# source code is translated into MSIL instructions, but how the .NET runtime uses a stack to hold values during execution. I'll leave the tracing of the second `for` loop to you as an exercise.

Quick Review

A stack is a specialized list whose elements are stored in last-in/first-out (LIFO) order. Stacks support two primary operations: push and pop. The push operation stores an item on top of the stack. As more items are pushed onto the stack, the older items move deeper into the stack while younger items are at the top of the stack. The most recent

item pushed onto the stack will always be at the top of the stack. The pop operation removes the most recently pushed item from the top of the stack.

A Home Grown Stack

In this section I want to show you how to use an array to implement a stack. Example 7.3 gives the code for a class I call HomeGrownStack.

7.3 HomeGrownStack.cs

```
1       using System;
2
3       public class HomeGrownStack {
4
5         private object[] stack_contents;
6         private int top = -1;
7         private const int INITIAL_SIZE = 25;
8
9         public HomeGrownStack(int initial_size){
10          stack_contents = new object[initial_size];
11        }
12
13        public HomeGrownStack():this(INITIAL_SIZE){ }
14
15        public bool IsEmpty {
16          get { return (top == -1); }
17        }
18
19        public void Push(object item){
20          if(item == null){
21            throw new ArgumentException("Cannot push null item onto stack!" );
22          }
23
24          if((++top) >= stack_contents.Length){
25            GrowStack();
26          }else{
27            stack_contents[top] = item;
28          }
29        } // end Push method
30
31        public object Pop(){
32          if(IsEmpty){
33            throw new InvalidOperationException("The stack is empty!");
34          }
35          object return_object = stack_contents[top];
36          stack_contents[top--] = null;
37          return return_object;
38        } // end Pop method
39
40        public object Peek(){
41          if(IsEmpty){
42            throw new InvalidOperationException("The stack is empty!");
43          }
44          return stack_contents[top];
45
46        } // end Peek method
47
48        private void GrowStack(){
49          object[] temp_array = new object[stack_contents.Length];
50          for(int i = 0; i < stack_contents.Length; i++){
51            temp_array[i] = stack_contents[i];
52          }
53
54          stack_contents = new object[stack_contents.Length * 2];
55
56          for(int i = 0; i < temp_array.Length; i++){
57            stack_contents[i] = temp_array[i];
58          }
59        } // end GrowArray method
60
61
62      } // end class definition
```

Referring to example 7.3 — the HomeGrownStack class contains three fields: an array of objects named stack_contents, an integer variable named top that points to the top of the stack, and a constant named INITIAL_SIZE which I have initialized to 25. On line 9 the constructor method takes one integer argument that sets

the size of the stack. It uses this parameter to create the object array. The default constructor on line 13 simply calls the first constructor while supplying the INITIAL_SIZE constant as an argument. On line 15 the IsEmpty property is defined. If the top field equals -1 it returns true, otherwise it returns false.

The Push() method definition starts on line 19. The first order of business is to ensure the incoming object reference is valid. If not, the method throws an ArgumentException. If the incoming reference is valid, the top variable is incremented and its value compared with the length of the array. If necessary, the array is dynamically grown to accommodate new items, otherwise there's enough room in the array to push the incoming reference which is assigned to the element pointed to by top.

The Pop() method begins on line 31. First, the method checks to see if the stack is empty. If so, an InvalidOperationException is thrown and the method exits. Otherwise, the element pointed to by top is returned, top is set to null and decremented by 1. The item removed from the array is returned and the method exits.

The Peek() method on line 40 throws an InvalidOperationException if the stack is empty, otherwise it returns a reference to the element on top of the stack but does not remove the element.

The GrowStack() method beginning on line 48 simply grows the array when the value of top approaches the value of the length of the array.

Example 7.4 gives a short program showing the HomeGrownStack in action. This short program reverses the order of a set of integers.

7.4 MainApp.cs (Demonstrating HomeGrownStack)

```
1        using System;
2
3        public class MainApp {
4          public static void Main(){
5            HomeGrownStack stack = new HomeGrownStack();
6            for(int i = 0; i < 37; i++){
7              stack.Push(i);
8            }
9
10           for(int i = 0; i < 37; i++){
11             Console.Write(stack.Pop() + " ");
12           }
13           Console.WriteLine();
14
15           // try one more Pop operation
16           try{
17             stack.Pop();
18            }catch(Exception e){
19              Console.WriteLine(e);
20            }
21
22         }
23       }
```

Referring to example 7.4 — an instance of HomeGrownStack is created on line 5. The `for` loop on line 6 pushes 38 integer values onto the stack. The `for` loop on line 10 pops each integer off the stack and writes its value to the console. This has the effect of reversing the sequence of integers generated by the `for` loop. Finally, one more call to the Pop() method is made inside of a `try/catch` block. The results of running this program appear in figure 7-3.

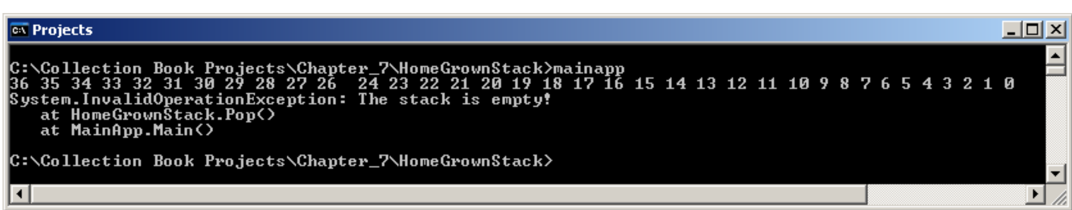


Figure 7-3: Results of Running Example 7.4

Quick Review

The HomeGrownStack class demonstrates the use of an array to contain stack items. It implements the Push(), Pop(), and Peek() methods as well as the IsEmpty property. The top field is incremented each time an item is pushed onto the stack and decremented each time an item is popped off the stack.

The Stack Class

In this section I discuss the non-generic Stack class which is found in the System.Collections namespace. I'll present its inheritance hierarchy and talk about some of the operations it supports in addition to the basic stack operations push and pop.

As a non-generic collection, the Stack class pushes and pops any type of object. Value type objects, as I demonstrated earlier, are boxed before being pushed onto the stack. When you pop an object off the stack you must cast it to its proper type. If it's a value type object it will undergo an unboxing operation as well.

Stack Class Inheritance Hierarchy

Figure 7-4 gives the UML class diagram for the System.Collections.Stack class inheritance hierarchy.

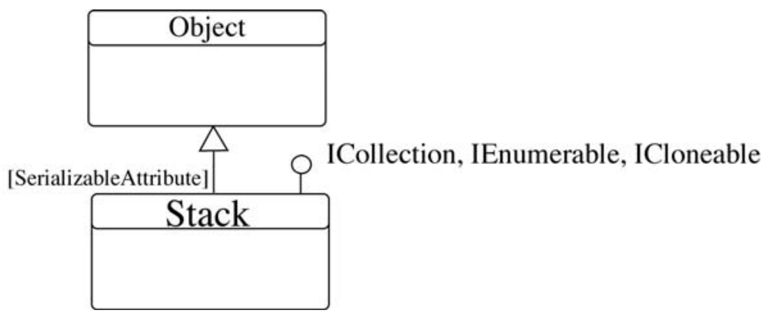


Figure 7-4: System.Collections.Stack Class Inheritance Hierarchy

Referring to figure 7-4 — the Stack class implicitly extends System.Object and implements the ICollection, IEnumerable, and ICloneable interfaces. It's also serializable.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the stack using the `foreach` statement. The direction of iteration begins with the stack's top element and ends with the oldest element on the stack.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the stack to an array. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which is discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the ICloneable Interface

The ICloneable interface exposes the Clone() method which is used to make a shallow copy of the stack.

Balanced Symbol Checker

The following example shows the Stack class in action. The BalancedSymbolChecker class implements logic to parse a sequence of characters and look for balanced sets of parenthesis `'('` `')'`, braces `'{'` `'}'`, and brackets `'['` `']'`.

7.5 BalancedSymbolChecker.cs

```
using System;
using System.Collections;

public class BalancedSymbolChecker {

  private const char OPEN_PAREN = '(';
  private const char CLOSE_PAREN = ')';
  private const char OPEN_BRACKET = '[';
  private const char CLOSE_BRACKET = ']';
  private const char OPEN_BRACE = '{';
  private const char CLOSE_BRACE = '}';


  public char GetNextSymbol(){
    char c;
    do {
      if((c = (char)Console.Read()) == '\r'){
        return '\0';
      }
    }while( (c != OPEN_PAREN) && (c != CLOSE_PAREN) && (c != OPEN_BRACKET) && (c != CLOSE_BRACKET)
            && (c != OPEN_BRACE) && (c != CLOSE_BRACE) );

      return c;
  } // end GetNextSymbol method

  public bool CheckMatch(char openSymbol, char closeSymbol){
    if((openSymbol == OPEN_PAREN) && (closeSymbol != CLOSE_PAREN) ||
       (openSymbol == OPEN_BRACKET) && (closeSymbol != CLOSE_BRACKET) ||
       (openSymbol == OPEN_BRACE) && (closeSymbol != CLOSE_BRACE) ) {
         Console.WriteLine("Open Symbol " + openSymbol + " does not match " + closeSymbol);
         return false;
       }
       return true;
  }

  public bool CheckBalance(){
    char c, match;
    int errors = 0;

    Stack pendingTokens = new Stack();
    while((c = GetNextSymbol()) != '\0'){
      switch(c){
        case OPEN_PAREN:
        case OPEN_BRACKET:
        case OPEN_BRACE: pendingTokens.Push(c);
                  break;
        case CLOSE_PAREN:
        case CLOSE_BRACKET:
        case CLOSE_BRACE: {
                    if(pendingTokens.Count == 0){
                      Console.WriteLine("Invalid symbol sequence: " + c);
                      return false;
                    }else{
                      match = (char)pendingTokens.Pop();
                      if(! CheckMatch(match, c)){
                       return false;
                      }else{
                        Console.WriteLine("Matching symbols {0} and {1} found", match, c);
                      }
                    }
                    break;
                  }

      }
    }

    while(pendingTokens.Count > 0){
       match = (char)  pendingTokens.Pop();
       Console.WriteLine("Unmatched symbol: " + match);
       errors++;
      }
      return (errors > 0) ? false:true;
  }
} // end BalancedSymbolChecker class definition
```

Referring to example 7.5 — the BalancedSymbolChecker class defines a set of constants that represent each of the six symbols of interest. It defines three methods: GetNextSymbol(), CheckMatch(), and CheckBalance(). The GetNextSymbol() method uses the Console.Read() method to read a line of text from the console. Each subsequent call to the Console.Read() method will return the next character from the line of text until it encounters the end-of-

line sequence. In this example I have used the carriage return `'\r'` character to signal the end of the character sequence. The GetNextSymbol() method ignores all characters other than one of the six symbols. When it encounters one of the six symbols it returns that symbol, otherwise it returns `'\0'` to signal it has reached the end of the character sequence.

The CheckMatch() method compares two symbols with each other. If they match it returns true; if not it returns false.

These two methods are used in the CheckBalance() method which begins on line 36. On line 40, a Stack named pendingTokens is used to hold symbols for future evaluation. Most of the action happens within the body of the `while` loop starting on line 41. While there is a symbol to evaluate it is presented to the `switch` statement on line 42. If it's an opening symbol it's pushed onto the stack. If it's a closing symbol and the pendingTokens.Count == 0 then it has encountered an invalid symbol sequence. If the pendingTokens stack contains symbols, the last symbol is popped off the stack and compared with the closing symbol. If they match, a message is written to the console displaying the matching symbols, if not, the method returns false.

Example 7.6 offers a short program demonstrating the use of the BalancedSymbolChecker class.

7.6 MainApp.cs (Demonstrating BalancedSymbolChecker)

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        BalancedSymbolChecker checker = new BalancedSymbolChecker();
6        char c = '\0';
7        while(((c = checker.GetNextSymbol()) != '\0')){
8          Console.Write(c + " ");
9        }
10       Console.WriteLine();
11       Console.WriteLine("---------------------");
12       checker.CheckBalance();
13     }
14   }
```

Referring to example 7.6 — an instance of BalancedSymbolChecker is created on line 5. The while loop uses the GetNextSymbol() method to read a line of characters from the console and print the extracted symbols to the console. On line 12 the CheckBalance() method is called, which will parse another line of characters and check the input symbols for balance. Figure 7-5 shows the results of running this program.

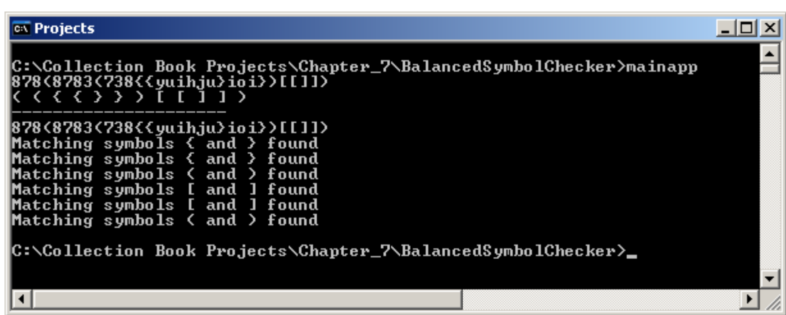


Figure 7-5: Results of Running Example 7.6

Quick Review

The System.Collections.Stack is a non-generic collection which stores any type of object. Value type objects will undergo a boxing operation when they are pushed onto the stack. This results in their references being pushed onto the stack and the object it points to is created in the heap. When the value type object is popped off the stack, it will undergo an unboxing operation.

The Stack<T> Class

The generic Stack<T> class is the direct replacement for the non-generic Stack class. It provides a lot more functionality in the form of extension methods defined by the System.Linq.Enumerable class. It also provides optimal performance when used with value type objects as they do not require boxing/unboxing operations when being pushed on and popped off the stack.

Stack<T> Class Inheritance Hierarchy

Figure 7-6 gives the UML class diagram showing the inheritance hierarchy of the Stack<T> class.

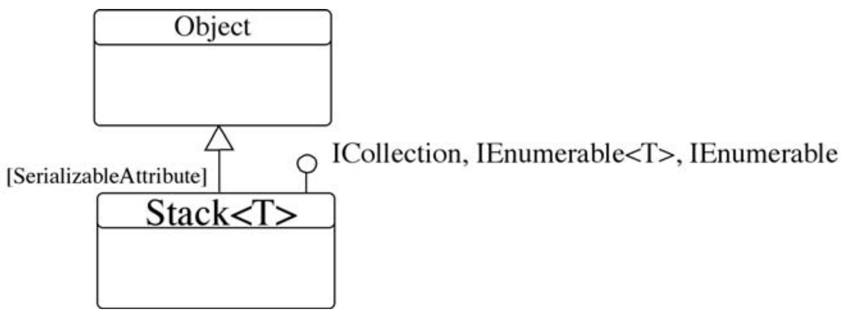


Figure 7-6: Stack<T> Class Inheritance Hierarchy

Referring to figure 7-6 — the Stack<T> class implicitly extends System.Object and implements the ICollection, IEnumerable, and IEnumerable<T> interfaces.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the stack using the `foreach` statement. The direction of iteration begins with the stack's top element and ends with the oldest element on the stack.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the stack to an array. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which is discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the IEnumerable<T> Interface

The IEnumerable<T> interface extends IEnumerable and allows the elements of the generic Stack<T> class to be enumerated by the `foreach` statement.

What Happened to ICollection<T>?

The Stack<T> class is one of two generic collection classes that do not explicitly implement the ICollection<T> interface, rather, it directly implements a few of its methods in the interest of providing specialized control over access to collection elements. For example, you can only add elements to a Stack<T> class via its Push() method, and only remove elements via the Pop() and Clear() methods.

Command Line PostFix Calculator

Example 7.7 shows a generic Stack<T> class in action in a program that implements a postfix command-line calculator. The LineCalc class provides addition, subtraction, multiplication, division, and exponent operations.

7.7 LineCalc.cs

```
using System;
using System.Collections.Generic;

public class LineCalc {

   private Stack<double> stack = new Stack<double>();
   private const char ADD = '+';
   private const char SUB = '-';
   private const char MULT = '*';
   private const char DIV = '/';
   private const char EXP = '^';
   private const char EQUALS = '=';


   public void ProcessLine(string input){

     try {
      double operand = Double.Parse(input);
      stack.Push(operand);

     } catch(Exception){
        this.ProcessOperator(input);
     }
   }

   public void ProcessOperator(string input){
     switch(input[0]){
       case ADD: Add();
                 break;

       case SUB: Sub();
                 break;

       case MULT: Mult();
                  break;

       case DIV:  Div();
                  break;

       case EXP:  Exp();
                  break;

       case EQUALS: Equals();
                    break;

       default: Console.WriteLine("Invalid Operator!");
               break;
     }
   }

   public void Add(){
     if(stack.Count >= 2){
      double operand_1 = stack.Pop();
      double operand_2 = stack.Pop();
      double result = operand_1 + operand_2;
      stack.Push(result);
      Console.WriteLine("Add result: {0}", result);
     } else{
       Console.WriteLine("Note enough operands on stack!");
     }
   }

   public void Sub(){
     if(stack.Count >= 2){
      double operand_1 = stack.Pop();
      double operand_2 = stack.Pop();
      double result = operand_2 - operand_1;
      stack.Push(result);
      Console.WriteLine("Sub result: {0}", result);
     } else{
       Console.WriteLine("Note enough operands on stack!");
     }
   }

```



```
        public void Mult(){
          if(stack.Count >= 2){
           double operand_1 = stack.Pop();
           double operand_2 = stack.Pop();
           double result = operand_1 * operand_2;
           stack.Push(result);
           Console.WriteLine("Mult result: {0}", result);
          }else{
            Console.WriteLine("Note enough operands on stack!");
          }
        }

        public void Div(){
          if(stack.Count >= 2){
           double operand_1 = stack.Pop();
           double operand_2 = stack.Pop();
           double result = operand_2 / operand_1;
           stack.Push(result);
           Console.WriteLine("Div result: {0}", result);
          }else{
            Console.WriteLine("Note enough operands on stack!");
          }
        }

        public void Exp(){
          if(stack.Count >= 2){
           double operand_1 = stack.Pop();
           double operand_2 = stack.Pop();
           double result = 1;
           for(int i = 0; i< operand_1; i++){
            result *= operand_2;
           }
           stack.Push(result);
           Console.WriteLine("Exp result: {0}", result);
          }else{
            Console.WriteLine("Note enough operands on stack!");
          }
        }

        public void Equals(){
          if(stack.Count >= 1){
           Console.WriteLine("Total: {0}", stack.Pop());
          }else{
            Console.WriteLine("Stack empty!");
          }
        }


        public static void Main(){
          LineCalc lc = new LineCalc();
          string input = String.Empty;
          Console.Write("Enter operand, operator, or \"quit\" to exit  --> ");
          while((input = Console.ReadLine()) != "quit"){
             if(input.Length > 0){
               lc.ProcessLine(input);
             }
             Console.Write("Enter operand, operator, or \"quit\" to exit --> ");

          }
        }
    }  // end LineCalc class definition
```

Referring to example 7.7 — the LineCalc program uses a generic stack of doubles (Stack<double>) to push and pop operands and the results of operations. The ProcessLine() method first assumes the input string is a valid double and tries to parse it as such. If the string fails to parse as a double an exception is thrown and it tries again to parse the string as an operator by calling the ProcessOperator() method in the body of the catch block. The ProcessOperator() method presents the first character of the input string (input[0]) to the switch statement. If the operator is one of the valid operators, the corresponding operation is performed. If not, an invalid operator message is written to the console and the program returns to waiting for valid input.

Note how the stack is used to store incoming operands and how, after each operation, the result is pushed back onto the top of the stack.

Entering the equals operator '=' results in the value located at the top of the stack being popped from the stack and written to the console. In this way you can clear the calculator of the last result before proceeding with a new calculation.

Figure 7-7 shows the LineCalc program in action.

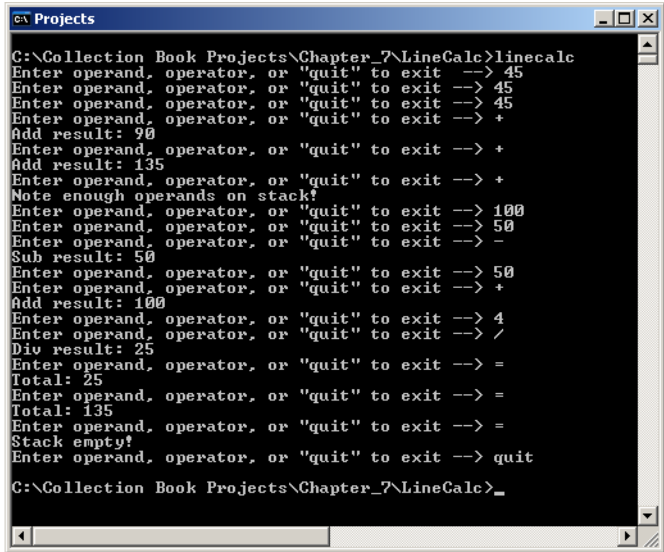


Figure 7-7: Results of Performing Several Operations with LineCalc

Quick Review

The System.Collections.Generic.Stack<T> class is the direct replacement for the non-generic Stack class. The benefit to using the Stack<T> class is that you gain a wider array of operations via extension methods defined by the System.Linq.Enumerable class. Also, value types do not require boxing and unboxing operations when being pushed onto and popped off the stack.

Summary

A stack is a specialized list whose elements are stored in last-in/first-out (LIFO) order. Stacks support two primary operations: push and pop. The push operation stores an item on top of the stack. As more items are pushed onto the stack, the older items move deeper into the stack while younger items are at the top of the stack. The most recent item pushed onto the stack will always be at the top of the stack. The pop operation removes the most recently pushed item from the top of the stack.

The HomeGrownStack class demonstrates the use of an array to contain stack items. It implements the Push(), Pop(), and Peek() methods as well as the IsEmpty property. The top field is incremented each time an item is pushed onto the stack and decremented each time an item is popped off the stack.

The System.Collections.Stack is a non-generic collection which stores any type of object. Value type objects will undergo a boxing operation when they are pushed onto the stack. This results in their references being pushed onto the stack and the object it points to is created in the heap. When the value type object is popped off the stack, it will undergo an unboxing operation.

The System.Collections.Generic.Stack<T> class is the direct replacement for the non-generic Stack class. The benefit to using the Stack<T> class is that you gain a wider array of operations via extension methods defined by the System.Linq.Enumerable class. Also, value types do not require boxing and unboxing operations when being pushed onto and popped off the stack.

References

Sten Henriksson. A Brief History of the Stack. [http://www.sigcis.org/files/A%20brief%20history.pdf]

Donald E. Knuth. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Third Edition. Addison-Wesley, Reading, Massachusetts. 1997. ISBN: 0-201-89683-4.

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

Chapter 8

Contax T

Train Queue

Queues

Learning Objectives

- *Describe the operation of a queue*
- *Describe the characteristics of First In/First Out (FIFO) processing*
- *List at least four examples of applications that utilize queues*
- *State the type of data structure used to implement the Queue and Queue<T> classes*
- *Describe what it means to enqueue an item into a queue*
- *Describe what it means to dequeue an item from a queue*
- *List and describe the members of the Queue and Queue<T> classes*
- *Use the non-generic Queue class in a program*
- *Use the generic Queue<T> class in a program*
- *Describe the functionality provided by each interface implemented by the Queue class*
- *Describe the functionality provided by each interface implemented by the Queue<T> class*

Introduction

Queues manifest themselves in many areas of our lives and are workhorse data structures in the areas of computers and computer science. Anytime you've waited in line for something, you've participated in queue operations. The first person to arrive in line at the bank is the first person to receive service from the next available teller. Likewise, most modern operating systems process events that have been waiting in some type of queue. (*See C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming, Chapter 12, for a discussion of the Microsoft Windows Message Queue*) In multitasking operating systems where executing threads are swapped in and out of the processor, waiting threads kick their heels in a queue until given another crack at the processor.

In this chapter I'll introduce you to the queue data structure. I'll show you how queues work and explain their characteristic operations: *enqueue* and *dequeue*. I'll then show you how a custom queue data structure might be implemented with the help of a *circular array*. Next, I'll demonstrate the use of the System.Collections.Queue and the System.Collections.Generic.Queue<T> classes.

When you finish reading this chapter you'll have a solid understanding of how queues work and understand when they're the right data structure to use in your programs.

Queue Operations

A queue is a list-based data structure whose elements are inserted at one end and removed from the other. This dual-ended operation gives the queue a First-In/First-Out (FIFO) characteristic.

Characteristic Queue Operations

If you've ever waited in line at the drive-thru you've participated in queuing operations: You arrive at one end of the line, wait your turn for service, and eventually emerge from the other end of the line when your turn at service arrives. This is how a queue operates.

Queue data structures support two primary operations: *enqueue* and *dequeue*.

Enqueue

Items are added to a queue with a call to the enque operation. Each call to enque increments the number of items in the queue by one.

Dequeue

Items are removed from a queue with a call to the dequeue operation. Each call to dequeue decrements the number of items in the queue by one.

An Illustration Will Help

Figure 8-1 shows a series of enqueue and dequeue operations being performed on a queue. Referring to figure 8-1 — the queue initially starts empty. Four elements are added to the queue with a series of four calls to the enqueue operation. The end of the queue increments to the next open position with each successive call to enqueue. The first call to dequeue removes the first item from the queue. The head of the queue increments by one with each successive call to dequeue. The queue is empty after the fourth call to dequeue.

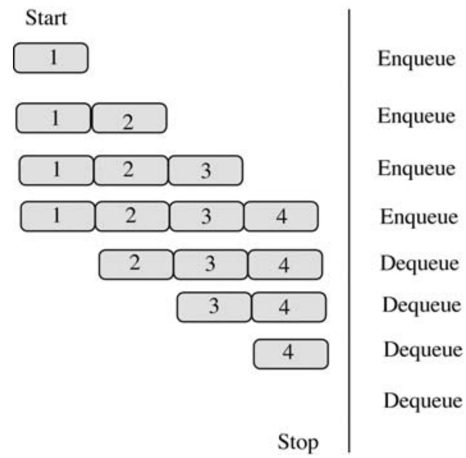


Figure 8-1: Enqueue and Dequeue Operations

Quick Review

Queues are work horse data structures in the real world as well as in the world of computers and computer science. Queues exhibit a First-In/First-Out (FIFO) characteristic; The first item to be inserted into a queue is the first item to be removed.

Queues support two primary operations: enqueue and dequeue. Items are added to a queue with a call to enqueue, and items are removed from a queue with a call to dequeue.

A Home Grown Queue Based on a Circular Array

In this section I want to show you how you might go about implementing a queue with the help of a *ring buffer* (a.k.a. circular buffer or circular array).

You can visualize a ring buffer as a circular list of elements in the shape of a ring as figure 8-2 illustrates.

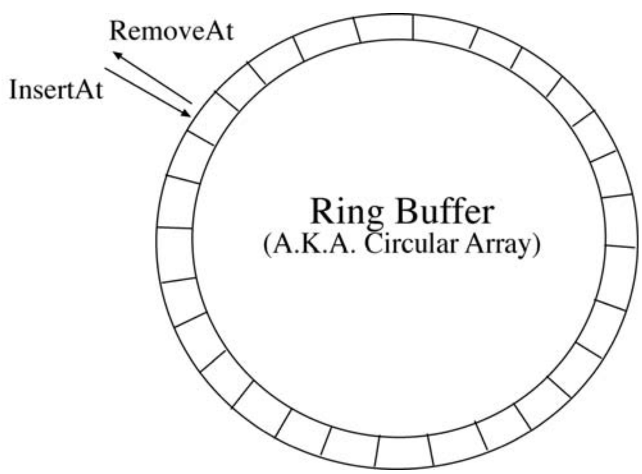


Figure 8-2: Empty Ring Buffer

Referring to figure 8-2 — initially, the ring buffer is empty and the InsertAt and RemoveAt indexes point to the same location. As elements are added to the ring buffer, the InsertAt index increments by one while the RemoveAt index remains unchanged as is shown in figure 8-3.

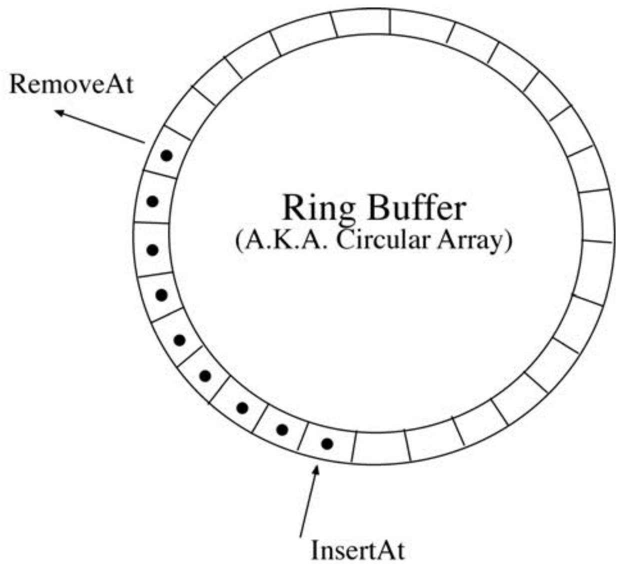


Figure 8-3: Ring Buffer After Items Have Been Inserted

In reality, memory is not circular and so a ring buffer must be implemented in terms of an ordinary array as figure 8-4 illustrates.

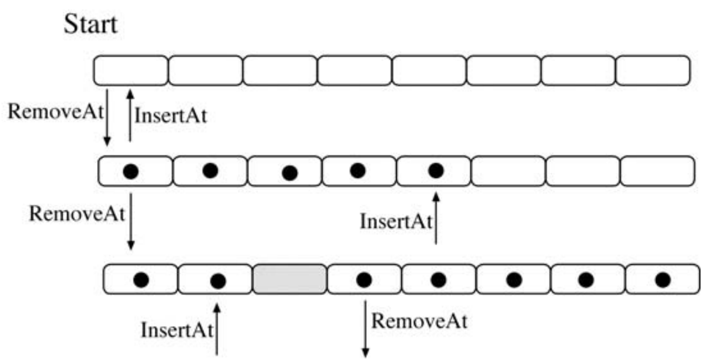


Figure 8-4: Ring Buffer Implemented with an Ordinary Array

Referring to figure 8-4 — an ordinary array has a limit to the number of elements it can hold. Initially, the array is empty and the InsertAt and RemoveAt indexes point to the same location. The insertion of elements into the array increments the InsertAt index while the RemoveAt index refers to the first item inserted into the array. When items are removed from the array, the RemoveAt index increments. If items have been removed from the array before the InsertAt index reaches the end of the array, it can be reset to point to the first element. This maximizes the use of space within the array. However, if no elements have been removed from the array by the time the InsertAt index reaches the end of the array, one of two things must happen: 1) either the insert operation must throw some type of exception indicating the array is full, or the array must be resized to accommodate additional elements. This is the approach taken with the CircularArray class given in example 8.1.

8.1 CircularArray.cs

```
using System;

public class CircularArray {
  private object[]  _array = null;
  private int _insertAt = 0;
  private int _removeAt = 0;
  private int _count = 0;
  private const int INITIAL_SIZE = 25;
  private bool _debug = true;

```



```
    public bool IsEmpty {
      get { return (_count == 0)?true:false; }
    }

    public int Count {
      get { return _count; }
    }

    public CircularArray(int initial_size, bool debug) {
      _array = new object[ initial_size];
      _debug = debug;
    }

    public CircularArray():this(INITIAL_SIZE, true){ }


    public void Insert(object item){
      if(item == null){
        throw new ArgumentException("Cannot insert null items!");
      }

      if((_insertAt >= _array.Length) && (_removeAt == 0)) { // we've inserted elements and removed none
        this.GrowArray();
      } else if((_insertAt >= _array.Length) && (_removeAt > 0)){ // There's room at the beginning
         _insertAt = 0; // reset
         _array[ _insertAt++] = item;
         _count++;
         return;
      } else if((_insertAt > 0) && (_insertAt == _removeAt)){ // Queue is full - grow and reorgansize
         this.GrowAndReorganizeArray();
      }
       _array[ _insertAt++] = item;
       _count++;
    } // end Insert() method


    public object Remove(){
      if(_count == 0){
        throw new InvalidOperationException("Array is empty!");
      }
      if(_removeAt >= _array.Length){
        _removeAt = 0;
      }
      object return_object = _array[ _removeAt];
      _array[ _removeAt++] = null;

      _count--;
      if((_count == 0) && (_removeAt == _insertAt)){ // reset insertion and removal points
        _removeAt = 0;
        _insertAt = 0;
      }
      return return_object;
    } // end Remove() method


    public object Peek(){
      if(_count == 0){
        throw new InvalidOperationException("Array is empty!");
      } else {
        return _array[ _removeAt];
      }
    } // end Peek() method


    private void GrowArray(){
      if(_debug){
        Console.WriteLine("-----Entering GrowArray Method------");
      }
      object[] temp_array = new object[ _array.Length];
      for(int i = 0; i < _array.Length; i++){
        temp_array[ i] = _array[ i];
      }

      _array = new object[ _array.Length * 2]; // double the size of the array

      for(int i = 0; i < temp_array.Length; i++){
        _array[ i] = temp_array[ i];
      }

      if(_debug){
        Console.WriteLine("-----Leaving GrowArray Method------");
```



```
92          }
93        } // end GrowArray() method
94
95        private void GrowAndReorganizeArray(){
96          if(_debug){
97            Console.WriteLine("-----Entering GrowAndReorganizeArray Method------");
98          }
99
100         object[ ] temp_array = new object[ _array.Length];
101         for(int i = 0; i < _array.Length; i++){
102           temp_array[ i] = _array[ i];
103         }
104
105         int old_length = _array.Length;
106
107         _array = new object[ old_length * 2]; // double the size of the array
108
109         int j = 0;
110         for(int i = _removeAt; i < old_length; i++){
111           _array[ j++] = temp_array[ i];
112         }
113
114         for(int i = 0; i < _insertAt; i++){
115           _array[ j++] = temp_array[ i];
116         }
117
118         _removeAt = 0;
119         _insertAt = _count;
120
121         if(_debug){
122           Console.WriteLine("-----Leaving GrowAndReorganizeArray Method------");
123         }
124       }
125
126
127     } // end CircularArray class definition
```

Referring to example 8.1 — an array of objects named _array serves as the basis for the circular array. The fields _insertAt, _removeAt, and _count are used to manage the internal state of the circular array. When a CircularArray object is created, its initial size can be specified via the constructor, or, if the default constructor is called, its initial size will be set to 25 elements.

The Insert() method starts on line 27 and checks first to ensure that inserted elements are not null. Next, the `if` statement on line 32 checks to see if any elements have been removed from the array. If not, the array is full and it must be expanded to hold more elements. This is accomplished with a call to the GrowArray() method.

If there’s room at the beginning of the array because the _removeAt index has been incremented, elements are inserted there. If, however, the _insertAt and _removeAt indexes are equal, then the array is full and must be expanded as well as reorganized before new elements can be added. This is accomplished with a call to the GrowAndReorganizeArray() method.

The Remove() method starts on line 47. If the array is empty, the method throws an InvalidOperationException, otherwise, the next object in the array is returned and the _removeAt index is incremented by one. When the value of _removeAt approaches the value of the length of the array, it’s reset to zero. If, after the removal of an element, the number of elements in the array equals zero, both the _insertAt and _removeAt indexes are reset to zero.

In this fashion, the CircularArray class expands its array to hold additional elements. When necessary, it can both expand and reorganize its array.

Example 8.2 gives the code for the HomeGrownQueue class whose functionality is based on the CircularArray class.

8.2 HomeGrownQueue.cs

```
using System;

public class HomeGrownQueue {
  private CircularArray _ca = null;
  private const int INITIAL_SIZE = 25;

  public HomeGrownQueue(int initial_size, bool debug){
    _ca = new CircularArray(initial_size, debug);
  }

  public HomeGrownQueue():this(INITIAL_SIZE, true){ }

  public bool IsEmpty {
    get { return _ca.IsEmpty; }
  }
```



```

   public int Count {
     get {  return _ca.Count; }
   }

   public void Enqueue(object item){
     try{
       _ca.Insert(item);
     } catch(Exception){
       Console.WriteLine("Cannot enqueue null item!");
     }
   }

   public object Dequeue(){
     object return_object = null;
     try{
      return_object = _ca.Remove();
     } catch(Exception){
       throw new InvalidOperationException("Queue is empty!");
     }
     return return_object;

   }

   public object Peek(){
     object return_object;
     try{
      return_object = _ca.Peek();
     } catch(Exception){
       throw new InvalidOperationException("Queue is empty!");
     }
     return return_object;
   }

 }
```

Referring to example 8.2 — the implementation of HomeGrownQueue is easy since the heavy lifting is done by the CircularArray class. The HomeGrownQueue implements a façade software design pattern and simply acts as a wrapper for the CircularArray class providing the Enqueue() and Dequeue() methods you expect from a queue. It also provides an IsEmpty property and a Peek() method, which in turn calls the CircularArray.Peek() method.

Example 8.3 gives the code for a MainApp class that puts the HomeGrownQueue through its paces.

8.3 MainApp.cs (Demonstrating HomeGrownQueue)

```
using System;

public class MainApp {
  public static void Main(){
   HomeGrownQueue queue = new HomeGrownQueue(); // default size of 25 elements

    for(int i = 0; i < 40; i++){  // test Growth Capability
      queue.Enqueue(i);
    }

    Console.WriteLine("Count = {0}", queue.Count);

    int itemsInQueue = queue.Count;

    Console.WriteLine("Next item to be removed from queue is: {0}", queue.Peek());

    for(int i = 0; i < itemsInQueue ; i++) {
      Console.Write(queue.Dequeue() + " ");
    }

    Console.WriteLine();

    try{
     queue.Dequeue(); // try to remove one more element

    } catch(Exception e){
      Console.WriteLine(e);
    }

   queue = new HomeGrownQueue(); // start again with 25 elements

   for(int i = 0; i < 23; i++){
     queue.Enqueue(i);
   }

   for(int i = 0; i < 10; i++){
     Console.Write(queue.Dequeue() + " ");
```



```
38          }
39          Console.WriteLine();
40
41          queue.Enqueue(23);
42          queue.Enqueue(24);
43          queue.Enqueue(25);
44          queue.Enqueue(26);
45          queue.Enqueue(27);
46          queue.Enqueue(28);
47          queue.Enqueue(29);
48          queue.Enqueue(30);
49          queue.Enqueue(31);
50          queue.Enqueue(32);
51          queue.Enqueue(33);
52          queue.Enqueue(34);
53          queue.Enqueue(35);
54          queue.Enqueue(36);
55          queue.Enqueue(37);
56          queue.Enqueue(38);
57          queue.Enqueue(39);
58          queue.Enqueue(40);
59          queue.Enqueue(41);
60          for(int i = 42; i < 134; i++){
61            queue.Enqueue(i);
62          }
63
64          for(int i = 0; i< 83; i++){
65            Console.Write(queue.Dequeue() + " ");
66          }
67
68          for(int i = 134; i < 289; i++){
69            queue.Enqueue(i);
70          }
71
72          Console.WriteLine("Count = " + queue.Count);
73
74          while(queue.Count > 0){
75            Console.Write(queue.Dequeue() + " ");
76          }
77
78          Console.WriteLine("Count = " + queue.Count);
79        }
80      }
```

Referring to example 8.3 — on line 5, an instance of HomeGrownQueue is created with the default constructor which creates an internal array in the CircularArray class with 25 elements. The `for` statement on line 7 tests the growth capability by creating and inserting 40 integers into the queue. The rest of the program enqueues and dequeues various numbers of integers to trigger both the growth and grow and reorganize capability. Figure 8-5 shows the results of running this program.

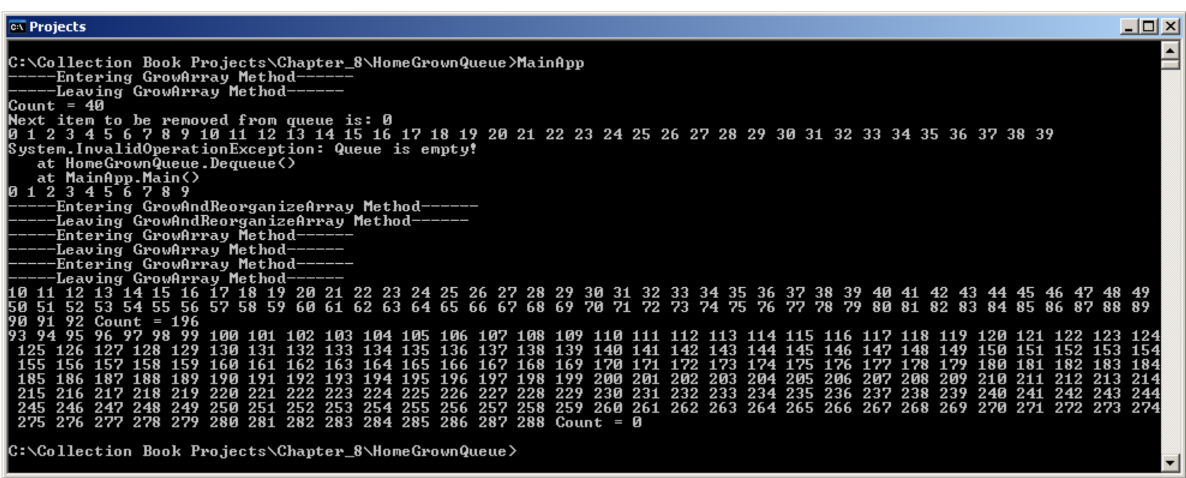


Figure 8-5: Results of Running Example 8.3

Quick Review

A ring buffer (a.k.a. circular array) can serve as the foundational data structure for a queue class. When implementing a circular array, you must carefully manage the insert and remove indexes to gain maximum space efficiency. The CircluarArray class implements an internal array growth capability as well as a grow and reorganize capability. The HomeGrownQueue class serves as a wrapper class for the CircularArray class.

The Queue Class

The System.Collections.Queue class is a non-generic collection class. It uses a circular array to implement queue functionality. And since it's a collection class, it has a lot more functionality that the HomeGrownQueue presented in the previous section.

Queue Class Inheritance Hierarchy

Figure 8-6 gives a UML class diagram of the Queue class.

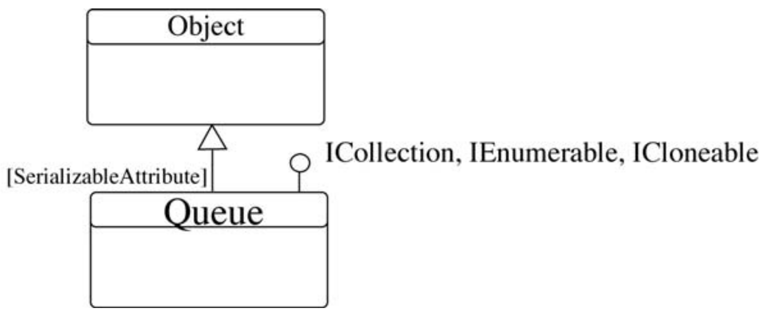


Figure 8-6: Queue Class Inheritance Hierarchy

Referring to figure 8-6 — the Queue class extends Object and implements ICollection, IEnumerable, and ICloneable. The following sections describe in more detail the purpose of each of these interfaces. It is also serializable.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the queue using the `foreach` statement. The direction of iteration begins with the queue's first, or oldest, element and ends with the last, or youngest, element inserted into the queue.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the queue to an array. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which are discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the ICloneable Interface

The ICloneable interface exposes the Clone() method which is used to make a shallow copy of the queue.

Palindrome Checker

A palindrome is a sequence of characters or numbers that can be read the same way in both directions. For example, the word “Eve” is a palindrome as is the phrase, “Madam, I’m Adam!” Example 8.4 gives the code for a program that uses both a queue and a stack to read a sequence of characters and determine if they form a palindrome.

8.4 PalindromeTester.cs

```
using System;
using System.Collections;

public class PalindromeTester {

  private int _letterCount;
  private int _checkedCharacters;
  private char _fromStack;
  private char _fromQueue;
  private bool _stillPalindrome;
  private Stack _stack;
  private Queue _queue;

  // Default constructor - Nothing to do, really
  public PalindromeTester(){ }

  public bool Test(String inputString){

    _stack = new Stack();
    _queue = new Queue();
    _letterCount = 0;

    foreach(char c in inputString){
      if(Char.IsLetter(c)){
        _letterCount++;
        char _c = Char.ToLower(c);
        _stack.Push(_c);
        _queue.Enqueue(_c);
      }
    }

    _stillPalindrome = true;
    _checkedCharacters = 0;
    while(_stillPalindrome && (_checkedCharacters < _letterCount)){
      _fromStack = (char)_stack.Pop();
      _fromQueue = (char)_queue.Dequeue();
      if(_fromStack != _fromQueue){
        _stillPalindrome = false;
      }
      _checkedCharacters++;
    }
    return _stillPalindrome;
  }

  public static void Main(){

     PalindromeTester pt = new PalindromeTester();
     string input_string = String.Empty;
     while(true){
       Console.Write("Please enter a possible palindrome for testing or \"Quit\" to exit: ");
       input_string = Console.ReadLine();

       if(input_string == "Quit") {
         return;
       }

       if(pt.Test(input_string)){
         Console.BackgroundColor = ConsoleColor.DarkBlue;
         Console.Beep(400, 600);
         Console.WriteLine("YES!!! \"{0}\" is a palindrome!", input_string);
         Console.BackgroundColor = ConsoleColor.Black;
         Console.ResetColor();
       }else{
         Console.BackgroundColor = ConsoleColor.Red;
         Console.Beep(78, 3000);
         Console.WriteLine("Sorry, \"{0}\" is not a palindrome...", input_string);
         Console.ResetColor();
       }
     }
  }
}
```


Referring to example 8.4 — the PalindromeTester class uses a queue and a stack, both from the System.Collections namespace, to test character sequences. The real work is performed by the Test() method, which begins on line 17. The method initializes the stack and the queue and sets the _letterCount field to 0. The `foreach` statement on line 23 iterates over the input string. If the character is a letter, it increments _letterCount by 1, converts it to lower case, and pushes it onto the stack. Characters that aren't letters are simply ignored. Remember, pushing the letters onto the stack has the effect of reversing the string.

The palindrome testing begins on line 32. The `while` statement on line 34 pops a character off the stack and dequeues a character from the queue and compares the two. If they match, it continues checking. If a match fails, then the sequence was not a palindrome. Note that when characters are popped off the stack and dequeued from the queue, they must be cast to their proper type before the comparison can be made.

The Main() method begins on line 45. It creates an instance of PalindromeTester and then executes the `while` loop on line 49 which repeatedly asks for input from users until they enter the string "Quit".

Figure 8-7 shows the results of running this program.

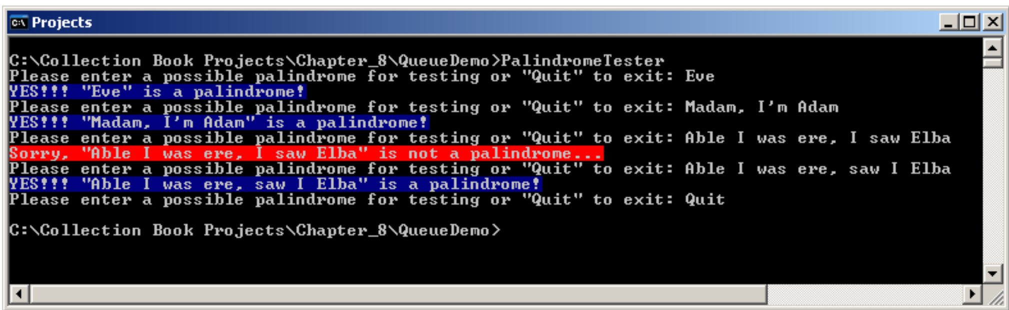


Figure 8-7: Results of Running Example 8.4

Referring to figure 8-7 — when the user enters a palindrome, the console background color is set to blue and a congratulatory message is written to the console. If the string is not a palindrome, the background color is set to red and the user receives the "Sorry..." message. How many palindromes can you think of?

Quick Review

A circular array is used to implement the System.Collections.Queue class. The Queue class extends System.Object and implements the IEnumerable, ICollection, and ICloneable interfaces. It is also serializable. Because it is a non-generic class, objects inserted into the queue must be cast to their proper type when they are dequeued.

The Queue<T> Class

The System.Collections.Generic.Queue<T> class is the generic version of the Queue class. The benefit to using the Queue<T> class is, among other things, not having to cast objects when they are dequeued.

Queue<T> Class Inheritance Hierarchy

Figure 8-8 gives the UML class diagram of the Queue<T> class inheritance hierarchy. Referring to figure 8-8 — the Queue<T> class extends System.Object and implements the ICollection, IEnumerable<T>, and IEnumerable interfaces. It is also serializable. It directly implements the ICollection<T> interface, just like the Stack<T> class does. The following sections describe the functionality provided by each of these interfaces.

Functionality Provided by the ICollection Interface

The ICollection interface inherits from IEnumerable and provides a CopyTo() method that can be used to copy the elements contained in the queue to an array. The ICollection interface also provides the Count, IsSynchronized,

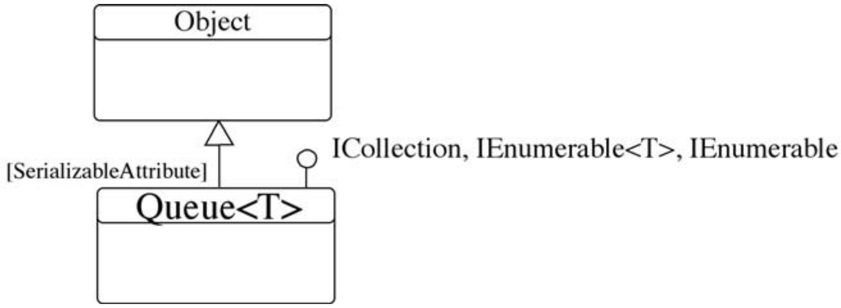


Figure 8-8: Queue<T> Class Inheritance Hierarchy

and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used in conjunction with multithreading programming techniques which is discussed in detail in Chapter 13 — Thread Programming.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface, along with the supporting IEnumerator interface, enables you to iterate over the elements in the queue using the `foreach` statement. The direction of iteration begins with the queue's last inserted, or oldest, element and ends with the most recently inserted, or youngest, element in the queue.

Functionality Provided by the IEnumerable<T> Interface

The IEnumerable<T> interface extends IEnumerable and allows the elements of the generic Queue<T> class to be enumerated by the `foreach` statement.

What Happened to ICollection<T>?

The Queue<T> class is one of two generic collection classes that do not explicitly implement the ICollection<T> interface, rather, it directly implements a few of its methods in the interest of providing specialized control over access to collection elements. For example, you can only add elements to a Queue<T> class via its Enqueue() method and only remove elements via the Dequeue() and Clear() methods.

Store Simulation

Queues come in handy when programming simulations of service scenarios. The code in example 8.5 uses a Queue<T> object to store DateTime objects in the simulation of a store checkout line with one checker. As with most simulations of this type, the primary purpose is to calculate the average and maximum waiting times.

8.5 StoreSimulation.cs

```
using System;
using System.Collections.Generic;

public class StoreSimulation {

  private Queue<DateTime> _queue;
  private TimeSpan _simulationRunTime;
  private int _totalServed;
  private TimeSpan _totalWaitingTime;
  private TimeSpan _maximumWaitingTime;
  private Random _rand;

  public StoreSimulation(int runtimeMinutes){
    _queue = new Queue<DateTime>();
    _simulationRunTime = new TimeSpan(0, runtimeMinutes, 0);
    _rand = new Random();
  }

  public StoreSimulation():this(10){}
```



```
20
21        public void Go(){
22          DateTime startTime = DateTime.Now;
23          Console.WriteLine("Simulation started at: {0}", startTime);
24          while((DateTime.Now - startTime) < _simulationRunTime){
25
26            if((_queue.Count > 0) && ((_rand.Next() % 6) == 3)){
27              DateTime t = _queue.Dequeue();
28              TimeSpan ts = DateTime.Now - t;
29              _totalServed++;
30              _totalWaitingTime += ts;
31              Console.Write("P");
32              if(_maximumWaitingTime < ts){
33                _maximumWaitingTime = ts;
34              }
35            }
36
37            switch(_rand.Next() % 4){
38              case 1 : _queue.Enqueue(DateTime.Now);
39                      Console.Beep();
40                      Console.ForegroundColor = ConsoleColor.Yellow;
41                      Console.Write(".");
42                      Console.ResetColor();
43                       break;
44              case 2 : _queue.Enqueue(DateTime.Now);
45                       _queue.Enqueue(DateTime.Now);
46                       Console.Beep(88, 200);
47                       Console.ForegroundColor = ConsoleColor.Blue;
48                       Console.Write("..");
49                       Console.ResetColor();
50                       break;
51              default: break;
52            }
53
54          }
55
56          // Print Statistics
57
58          Console.WriteLine();
59          Console.WriteLine("--------Simulation Complete ------------");
60          Console.WriteLine("Simulation ended at: {0}", DateTime.Now);
61          Console.WriteLine("Customers served: {0}", _totalServed);
62          Console.WriteLine("Average wait time: {0}", (double)_totalWaitingTime.Minutes/_totalServed);
63          Console.WriteLine("Longest wait time: {0}", _maximumWaitingTime);
64          Console.WriteLine("Customers still in line: {0}", _queue.Count);
65        }
66
67        public static void Main(){
68          StoreSimulation ss = new StoreSimulation(5);
69          ss.Go();
70        }  // end Main method
71      }  // End StoreSimulation Class Definition
```

Referring to example 8.5 — the bulk of the processing takes place inside the Go() method, which begins on line 21. The simulation will run for the amount of time specified in minutes via the constructor. The Go() method writes the simulation start time to the console. The `while` statement beginning on line 24 processes the simulation for the duration of the runtime. If the queue contains waiting objects, and the checker is free (determined with the calculation (_rand.Next() % 6) == 3)), a DateTime object is dequeued and processed.

The `switch` statement on line 37 is used to generate new "customers" either one at a time or two at a time. To track the generation and processing of customers I set the console foreground color to different colors to signify different generation and processing events. This makes the simulation easier and more fun to follow as it runs.

Finally, the short Main() method on line 67 creates an instance of StoreSimulation with a simulation runtime of 5 minutes and calls the Go() method to start processing. Figure 8-9 shows the results of running this program.

Quick Review

The Queue<T> class extends System.Object and implements the IEnumerable, IEnumerable<T>, and ICollection interfaces. It implements the ICollection<T> interface directly to limit insertion of objects into the queue via the Enqueue() method and the removal of objects via the Dequeue() and Clear() methods.

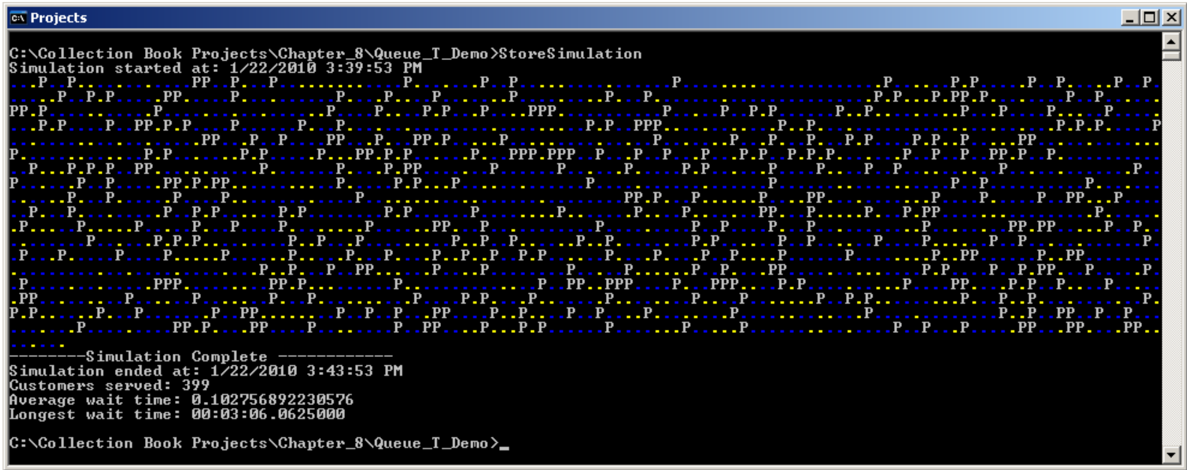


Figure 8-9: Results of Running Example 8.5

Summary

Queues are work horse data structures in the real world as well as in the world of computers and computer science. Queues exhibit a First-In/First-Out (FIFO) characteristic; The first item to be inserted into a queue is the first item to be removed.

Queues support two primary operations: enqueue and dequeue. Items are added to a queue with a call to enqueue, and items are removed from a queue with a call to dequeue.

A ring buffer (a.k.a. circular array) can serve as the foundational data structure for a queue class. When implementing a circular array, you must carefully manage the insert and remove indexes to gain maximum space efficiency. The CircluarArray class implements an internal array growth capability as well as a grow and reorganize capability. The HomeGrownQueue class serves as a wrapper class for the CircularArray class.

A circular array is used to implement the System.Collections.Queue class. The Queue class extends System.Object and implements the IEnumerable, ICollection, and ICloneable interfaces. It is also serializable. Because it is a non-generic class, objects inserted into the queue must be cast to their proper type when they are dequeued.

The Queue<T> class extends System.Object and implements the IEnumerable, IEnumerable<T>, and ICollection interfaces. It implements the ICollection<T> interface directly to limit insertion of objects into the queue via the Enqueue() method and the removal of objects via the Dequeue() and Clear() methods.

References

Donald E. Knuth. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Third Edition. Addison-Wesley, Reading, Massachusetts. 1997. ISBN: 0-201-89683-4.

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

Chapter 9

Contax T

Concrete Stairs

HashTables and Dictionaries

Learning Objectives

- Describe the purpose and use of a hashtable
- Describe the purpose and use of a dictionary
- Describe the purpose and use of key/value pairs
- Describe the purpose of a key
- List and describe the interfaces a class must implement if it's to be used as a key
- Describe the functionality provided by the IDictionary interface
- Use the Hashtable class in a program
- Use the Dictionary<TKey, TValue> class in a program
- Extract the key collection from a Hashtable or Dictionary
- Extract the value collection from a Hashtable or Dictionary

Introduction

Many times in your programming career you'll want to store an object in a collection and access that object via a unique *key*. This is the purpose of hashtables and dictionaries. Both hashtables and dictionaries perform the same type of service storing key/value pairs, but function differently on the inside. The Hashtable is the non-generic class and Dictionary<TKey, TValue> is its generic replacement.

In this chapter I'm going to explain how hashtables work. Along the way I'll introduce you to concepts like *buckets*, *hash functions*, *keys*, *values*, *collisions*, and *growth factors*. I'll start the discussion with an overview of how hashtables work followed by a detailed demonstration of the operation of a chained hashtable with the help of a comprehensive coding example. In the end, you'll be able to create your own hashtable class, but that's the beauty of the .NET collections framework. You won't have to!

A few topics I'd like to cover in this chapter I'm going to put off until Chapter 10 — Coding For Collections. These include how to create immutable objects and how to write a custom comparer. You don't need to know these specialized topics just yet to fully use the Hashtable or Dictionary<TKey, TValue> classes. You will, however, need to read chapter 10 before you can use your own custom developed classes as hashtable or dictionary keys.

So, in the absence of violent objection, let's get started.

Hashtable Operations

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1). Figure 9-1 offers a simple illustration of a hashtable and a hash function being applied to a key.

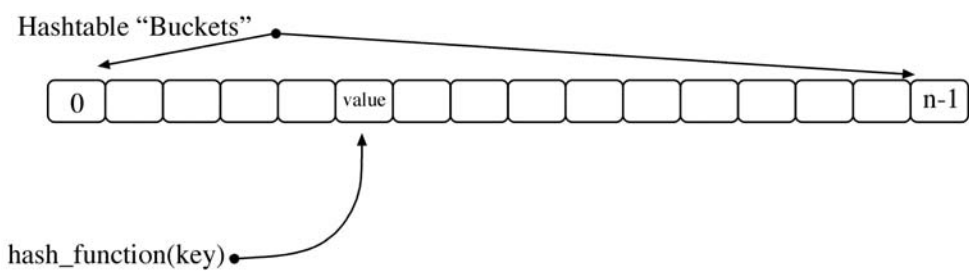


Figure 9-1: Typical Hashtable Showing Hash Function Being Applied to a Key

Referring to figure 9-1 — the hashtable elements are referred to as *buckets*. The hash function transforms the key into an integer value that falls within the bounds of the array. Into this location the key's corresponding object is placed. All subsequent hashtable accesses using that particular key will "hash" to the same location.

Hashtable Collisions

A potential problem with hashtables arises when the hash function calculates the same hash value for two different keys. When this happens a *collision* is said to have occurred. There are several ways to resolve hashtable collisions and their complete treatment here is beyond the scope of this book, but if you're interested, I recommend you refer to Donald Knuth's excellent treatment of the subject. (Donald Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching, Second Edition)

I will, however, discuss and demonstrate one collision resolution strategy referred to as *chaining*. Chaining can be used to resolve hashtable collisions by storing values whose keys have hashed to the same bucket in a chain of elements. See figure 9-2.

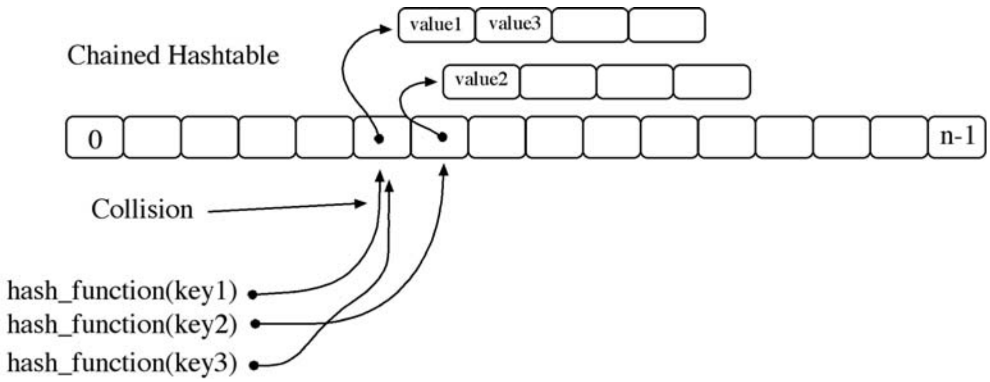


Figure 9-2: Collision Resolution within a Chained Hashtable

Referring to figure 9-2 — collisions occur when two different keys hash to the same bucket location. The corresponding values are stored in a linked list or similar structure.

Collisions can be kept to a minimum if the hash function is especially good, meaning that for a large number of different keys the resulting hash values are uniformly distributed over the range of available bucket values. Also, the size of the hashtable can be increased as the number of keys grows to decrease the likelihood of a collision. These concepts are demonstrated in the following section.

HomeGrownHashTable

The HomeGrownHashtable class presented in this section implements a complete example of a chained hashtable. The complete example consists of the KeyValuePair structure, presented in example 9.1, and the HomeGrownHashtable class, which is presented in example 9.2.

9.1 KeyValuePair.cs

```
using System;

public struct KeyValuePair : IComparable<KeyValuePair>, IComparable<string> {

  private string _key;
  private string _value;

  public KeyValuePair(string key, string value){
    _key = key;
    _value = value;
  }

  public string Key {
    get {  return _key; }
    set {  _key = value; }
  }

  public string Value {
    get {  return _value; }
    set {  _value = value; }
  }

  public int CompareTo(KeyValuePair other){
    return this._key.CompareTo(other.Key);
  }

  public int CompareTo(string other){
    return this._key.CompareTo(other);
  }
} // End KeyValuePair class definition
```

Referring to example 9.1 — the KeyValuePair structure represents the key/value pair that will be inserted into the hashtable. It defines two private fields named _key and _value and their corresponding public properties. The KeyValuePair structure also implements two interfaces: IComparable<KeyValuePair> and IComparable<string>. It implements these interfaces by defining two methods named CompareTo(KeyValuePair other) and CompareTo(string other). The implementation of the IComparable interfaces is necessary if KeyValuePair objects are to be used in sorting and searching operations. For a more detailed discussion on how to prepare your custom structures and classes for use in collections see Chapter 10 — Coding for Collections.

Example 9.2 lists the code for the HomeGrownHashtable class. It uses the KeyValuePair structure as part of its implementation.

9.2 HomeGrownHashtable.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

public class HomeGrownHashtable {

  private float _loadFactor = 0;
  private List<string> _keys = null;
  private List<KeyValuePair>[] _table = null;
  private int _tableSize = 0;
  private int _loadLimit = 0;
  private int _count = 0;

  private const float MIN_LOAD_FACTOR = .65F;
  private const float MAX_LOAD_FACTOR = 1.0F;


  // Constructor methods
  public HomeGrownHashtable(float loadFactor, int initialSize){
    if((loadFactor < MIN_LOAD_FACTOR) || (loadFactor > MAX_LOAD_FACTOR)){
      Console.WriteLine("Load factor must be between {0} and {1}." +
                        "Load factor adjusted to {1}", MIN_LOAD_FACTOR, MAX_LOAD_FACTOR);
      _loadFactor = MAX_LOAD_FACTOR;
    } else {
      _loadFactor = loadFactor;
      }

    _keys = new List<string>();
    _tableSize = this.DoublePrime(initialSize/2);
    _table = new List<KeyValuePair>[_tableSize];
    _loadLimit = (int)Math.Ceiling(_tableSize * _loadFactor);

    for(int i = 0; i<_table.Length; i++){
      _table[i] = new List<KeyValuePair>();
    }
  } // end constructor


  public HomeGrownHashtable():this(MAX_LOAD_FACTOR, 6){ }


  public string[] Keys {
    get { return _keys.ToArray(); }
  } // end Keys property


  public void Add(string key, string value){
    if((key == null) || (value == null)){
      throw new ArgumentException("Key and Value cannot be null.");
    }

    string upperCaseKey = key.ToUpper();

    if(_keys.Contains(upperCaseKey)){
      throw new ArgumentException("No duplicate keys allowed in HomeGrownHashtable!");
    }

    _keys.Add(upperCaseKey);
    _keys.Sort();
    int hashValue = this.GetHashValue(upperCaseKey);
    KeyValuePair item = new KeyValuePair(upperCaseKey, value);
    _table[hashValue].Add(item);
    _table[hashValue].Sort();

    if((++_count) >= _loadLimit){
      this.GrowTable();
    }
  }// end Add() method


  public string Remove(string key){
    if(key == null){
      throw new ArgumentException("Key string cannot be null!");
    }

```



```
      string upperCaseKey = key.ToUpper();

      if(_keys.Contains(upperCaseKey)){
        _keys.Remove(upperCaseKey);
      } else {
        throw new ArgumentException("Key does not exist in table!");
      }

      _keys.Sort();
      int hashValue = this.GetHashValue(upperCaseKey);
      string return_value = string.Empty;
      for(int i = 0; i<_table[ hashValue] .Count; i++){
        if(_table[ hashValue][ i] .Key == upperCaseKey){
          return_value = _table[ hashValue][ i] .Value;
          _table[ hashValue] .RemoveAt(i);
          _table[ hashValue] .Sort();
          break;
        }
      }

      return return_value;
    } // end Remove() method


    private void GrowTable(){
      List<KeyValuePair>[] temp = new List<KeyValuePair>[ _table.Length] ;
      for(int i=0; i<_table.Length; i++){
        temp[ i] = _table[ i] ;
      }

      _table = new List<KeyValuePair>[ this.DoublePrime(_tableSize)] ;

      for(int i=0; i<temp.Length; i++){
        _table[ i] = temp[ i] ;
      }

      for(int i=temp.Length; i<_table.Length; i++){
        _table[ i] = new List<KeyValuePair>();
      }
    } // end GrowTable() method


    public string this[ string key]{
      get {
            if((key == null) || (key == string.Empty)){
              throw new ArgumentException("Index key value cannot be null or empty!");
            }
            string return_value = string.Empty;
            string upperCaseKey = key.ToUpper();
            if(_keys.Contains(upperCaseKey)){
              int hashValue = this.GetHashValue(upperCaseKey);
              for(int i = 0; i<_table[ hashValue] .Count; i++){
                if(_table[ hashValue][ i] .Key == upperCaseKey){
                  return_value = _table[ hashValue][ i] .Value;
                  break;
                }
              }
            }
            return return_value;
          }

      set {
           if((key == null) || (key == string.Empty)){
              throw new ArgumentException("Index key value cannot be null or empty!");
            }

            if((value == null) || (value == string.Empty)){
              throw new ArgumentException("String value cannot be null or empty!");
            }
           string upperCaseKey = key.ToUpper();
            if(_keys.Contains(upperCaseKey)){
              int hashValue = this.GetHashValue(upperCaseKey);
              for(int i = 0; i<_table[ hashValue] .Count; i++){
                if(_table[ hashValue][ i] .Key == upperCaseKey){
                 KeyValuePair kvp = new KeyValuePair(upperCaseKey, value);
                 _table[ hashValue] .RemoveAt(i);
                 _table[ hashValue] .Add(kvp);
                 _table[ hashValue] .Sort();
                  break;
                }
              }
```



```
            }
      }
    } // end indexer


    private int DoublePrime(int currentPrime){
      currentPrime *= 2;
      int limit = 0;
      bool prime = false;
      while(!prime){
        currentPrime++;
        prime = true;
        limit = (int)Math.Sqrt(currentPrime);
        for(int i = 2; i<=limit; i++){
          if((currentPrime % i) == 0){
          prime = false;
          break;
          }
        }
      }
      return currentPrime;
    } // end DoublePrime() method


    private int GetHashValue(string key){
      int hashValue = ( Math.Abs(key.GetHashCode()) % _tableSize);
      return hashValue;
    } // end GetHashValue() method


    public void DumpContentsToScreen(){
      foreach(List<KeyValuePair> element in _table){
        foreach(KeyValuePair kvp in element){
          Console.Write(kvp.Value + " ");
        }
        Console.WriteLine();
      }
    } // end DumpContentsToScreen() method
  } // end class definition
```

Referring to example 9.2 — the HomeGrownHashtable contains a List<string> object named _keys into which incoming key values are insert for future reference, and for the main table, a List<KeyValuePair> array named _table. The other fields include _tableSize, _loadFactor, _loadLimit, and _count. The *load factor* is used to calculate the *load limit*. In HomeGrownHashtable, the load factor is allowed to range between .65 and 1. When items are inserted into the hashtable, the calculated load limit is compared to the item count and if necessary, the main table is expanded to hold more elements.

The HomeGrownHashtable class defines the following methods: two constructors — one that does all the heavy lifting and a default constructor; Add(), Remove(), GrowTable(), GetHashValue(), DoublePrime(), and DumpContentsToScreen(). It also defines a Keys property and an indexer which allows values to be retrieved via their keys using familiar array notation. (i.e. Hashtable_Reference[“key”])

Let’s step through the operation of the Add() method. The Add() method takes two arguments: a key string and a value string. The incoming arguments are checked for null values and if either are null the method throws an ArgumentException. The incoming key is converted to upper case with the String.ToUpper() method. The method then searches the _keys list to see if the key has already been inserted into the hashtable. If so, no duplicate keys are allowed and the method throws an ArgumentException. If the key is not in the _keys list, it’s added to the list and the _keys list is then sorted. The key is then used to generate a hash value with the help of the GetHashValue() method. A new KeyValuePair object is created and added to the list at the _table[hashValue] location. That list is then sorted. The _count field is incremented and if necessary, the _table is expanded to hold additional elements by a call to the GrowTable() method.

Let’s now examine the GrowTable() method. As its name implies, the purpose of the GrowTable() method is to grow the main hashtable (_table) to accommodate additional elements. The table growth mechanism is triggered in the Add() method when the element count (_count) approaches the hashtable’s calculated load limit. The load limit (_loadLimit) is calculated in the body of the constructor: _loadLimit = (int)Math.Ceiling(_tableSize * _loadFactor); The first order of business in the GrowTable() method is to create a temporary List<KeyValuePair> array named temp and copy all the existing elements from _table to temp. A new array of List<KeyValuePair> elements is created double the size of the existing table rounded up to the nearest prime number. This is done with the help of the DoublePrime() method. The reason I did this was because in my initial version of this example I used a custom hash function which relied on the generation of prime numbers to calculate the hash value of the key. I left the DoublePrime()

method in the code so you can experiment with different hash generation techniques, most of which rely on prime numbers. (Note: The DoublePrime() method replaces the usual approach of maintaining an array of precalculated prime numbers.)

The GetHashValue() method calculates a hash value based on the key. Since I'm using strings as keys, I decided to rely on the GetHashCode() method defined by the String class. This value is then modded (%) with _tableSize to yield a value between 0 and _tableSize - 1. You can experiment with different hash generation formulas by replacing key.GetHashCode() with a custom hash generation function.

The indexer, which starts on line 119, allows values stored within HomeGrownHashtable to be accessed and set with familiar array notation using the key. It consists of two parts: the `get` and `set` sections. The `get` section checks the key to ensure its not null or the empty string. The key is converted to upper case and its existence is checked in the _keys list. If it's in the list, a hash value is generated and used to find the value's location within the _table. The KeyValuePair list located at that location must then be searched to find the key. When the key is found, the corresponding value is used to set return_value.

The `set` section works similar to the `get` section except that when the key is located, that KeyValuePair is removed and a new one created and added to the KeyValuePair list at that table location. The list is then sorted.

Example 9.3 offers a MainApp class that demonstrates the use of HomeGrownHashtable.

9.3 MainApp.cs (Demonstrating HomeGrownHashtable)

```
1    using System;
2
3    public class MainApp {
4      public static void Main(string[] args){
5        HomeGrownHashtable ht = new HomeGrownHashtable();
6        ht.Add("Rick", "Photographer, writer, publisher, handsome cuss");
7        ht.Add("Coralie", "Gorgeous, smart, funny, gal pal");
8        ht.Add("Kyle", "Tall, giant of a man! And a recent college graduate!");
9        ht.Add("Tati", "Thai hot sauce!");
10       Console.WriteLine(ht["Tati"]);
11       Console.WriteLine(ht["Kyle"]);
12       ht["Tati"] = "And a great cook, too!";
13       ht.DumpContentsToScreen();
14       ht.Remove("Tati");
15       ht.DumpContentsToScreen();
16     } // end Main() method
17   }
```

Referring to example 9.3 — an instance of HomeGrownHashtable is created on line 5. Lines 6 through 9 add several key/value pairs to hashtable. Lines 10 and 11 demonstrate the use of the indexer to access the values associated with the keys "Tati" and "Kyle". On line 12, the indexer is used to replace the value associated with the key "Tati" with a new value. On line 13 the DumpContentsToScreen() method is called followed by the removal of the item referred to by the key "Tati" from the hashtable. The DumpContentsToScreen() method is then called one last time. Figure 9-3 shows the results of running this program.

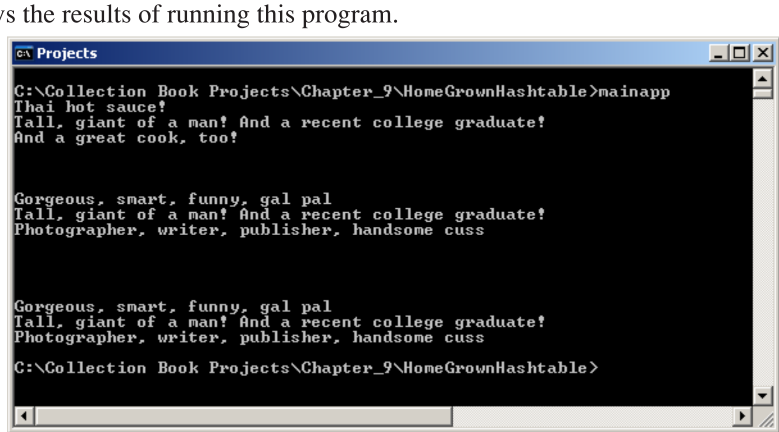


Figure 9-3: Results of Running Example 9.3

Quick Review

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1).

The HomeGrownHashtable class implements a chained hashtable where each hashtable *bucket* points to a List<KeyValuePair> object into which KeyValuePair objects are inserted. The hashtable's load limit determines when the hashtable should be grown to accommodate additional elements. The load limit is calculated by multiplying the table size by the load factor.

Hashtable *collisions* can occur when two different keys hash to the same hash value. The chained hash table resolves collisions by allowing collisions to occur and storing the KeyValuePair objects in a list at that location which must then be searched to find the key/value pair of interest.

Hashtable Class

The Hashtable class, located in the System.Collections namespace, stores hashtable elements as DictionaryEntry objects. A DictionaryEntry object consists of Key and Value properties and methods inherited from the System.Object class.

Unlike my HomeGrownHashtable discussed in the previous section, the Hashtable class doesn't use chaining to resolve collisions. According to Microsoft's documentation it uses a technique called *double hashing*. Double hashing works like this: If a key hashes to a bucket value already occupied by another key, the hash function is altered slightly and the key is rehashed. If that bucket location is empty the value is stored there, if it's occupied, the key must again be rehashed until an empty location is found.

Figure 9-4 shows the UML class diagram for the HashTable inheritance hierarchy.

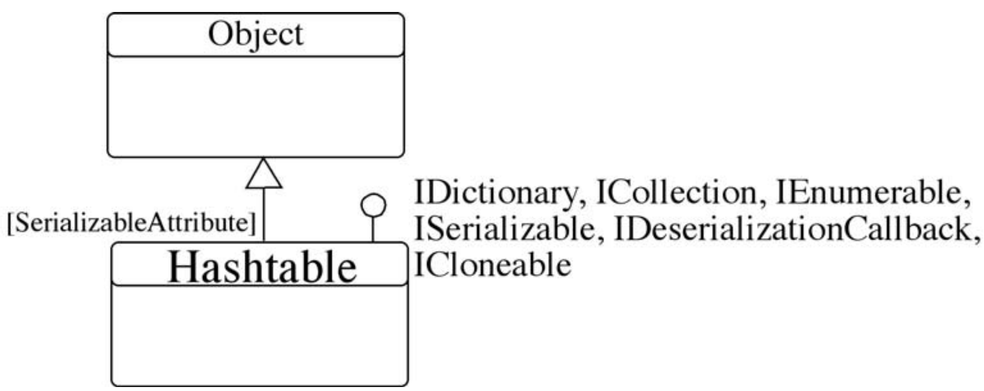


Figure 9-4: Hashtable Class Inheritance Hierarchy

Referring to figure 9-4 — The Hashtable class implements the IDictionary, ICollection, IEnumerable, ISerializable, IDeserializableCallback, and ICloneable interfaces. The functionality provided by each of these interfaces is discussed in more detail below.

Functionality Provided by the IEnumerable Interface

The IEnumerable interface allows the items contained within the Hashtable collection to be iterated over with the `foreach` statement. Note that each element of a Hashtable collection is a DictionaryEntry object. The code to iterate over each element of a Hashtable with a `foreach` statement would look something like the following code snippet, assuming there exists a reference to a Hashtable object named `ht`:

```
foreach(DictionaryEntry item in ht){
  Console.WriteLine(item.Key);
}
```

The code snippet above would print out the value of each key to the console.

The Hashtable class also provides several properties, two of which are Keys and Values. The Keys property returns an ICollection of the Hashtable's keys and the Values property returns an ICollection of the Hashtable's Values. You can use a `foreach` statement to step through each of these collections as the following code snippet demonstrates, assuming that keys are strings:

```
foreach(string key in ht.Keys){
  Console.WriteLine(key);
}
```

In this example, each key in the Keys collection is written to the console. Later, in the Hashtable example program, I'll show you another way to step through the items in a Hashtable using its indexer.

Remember — when stepping through a collection with the `foreach` statement, you can't modify the elements.

Functionality Provided by the ICollection Interface

The ICollection interface declares the GetEnumerator() and CopyTo() methods as well as the IsSynchronized and SyncRoot properties. The IsSynchronized and SyncRoot properties are discussed in greater detail in Chapter 14 — Collections and Threads.

Functionality Provided by the IDictionary Interface

The IDictionary interface declares the Add(), Remove(), and Contains() methods, the indexer (shown as Item in the properties list), and the Keys and Values properties.

Functionality Provided by the ISerializable and IDeserializationCallBack Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the Hashtable class requires custom serialization over and above simply tagging the class definition with the Serializable attribute. Custom serialization and deserialization is discussed in detail in Chapter 17 — Collections and I/O.

Functionality Provided by the ICloneable Interface

The ICloneable interface makes it possible to make copies of Hashtable objects.

Hashtable In Action

Example 9.4 demonstrates the use of the Hashtable collection. In this example, the program reads a text file line-by-line and stores each line in the Hashtable collection using the line number as the key. The name of the text file must be supplied on the command line when the program is executed.

9.4 HashtableDemo.cs

```
using System;
using System.Collections;
using System.IO;
using System.Text;


public class HashtableDemo {

  public static void Main(string[] args){
    FileStream fs = null;
    StreamReader reader = null;
    Hashtable ht = new Hashtable();
    try {
       fs = new FileStream(args[0], FileMode.Open);
       reader = new StreamReader(fs);

       int line_count = 1;
       string input_line = string.Empty;
       while((input_line = reader.ReadLine()) != null){
         string line_number_string = (line_count++).ToString();
         if(!ht.Contains(line_number_string)){
           ht.Add(line_number_string, input_line);
```



```
23              }
24            }
25
26            }catch(IndexOutOfRangeException){
27              Console.WriteLine("Please enter the name of a text file on the command line " +
28                                  "when running the program!");
29            }catch(Exception e){
30               Console.WriteLine(e);
31            }  finally {
32              if(fs != null){
33                fs.Close();
34              }
35              if(reader != null){
36                reader.Close();
37              }
38            }
39
40
41           for(int i = 1; i<=ht.Keys.Count; i++){
42             Console.WriteLine("Line {0}: {1}", i, ht[i.ToString()]);
43           }
44
45           Console.WriteLine("*********************************************");
46           Console.WriteLine("Line {0}: {1}", 2567, ht[2567.ToString()]);
47           Console.WriteLine("Line {0}: {1}", 193, ht[193.ToString()]);
48           Console.WriteLine("Line {0}: {1}", 669, ht[669.ToString()]);
49           Console.WriteLine("Line {0}: {1}", 733, ht[733.ToString()]);
50
51       } // end Main() method
52     } // end HashtableDemo class
```

Referring to example 9.4 — a FileStream reference named `fs` is declared on line 10 and initialized to null. On line 11 a StreamReader reference named `reader` is declared and also initialized to null. In the body of the try/catch block, which begins on line 13, the FileStream object is created using the first command line argument (args[0]). The FileStream object is then used to create the StreamReader object on the following line. On line 17, a local variable named `line_count` is declared and initialized to 1. An `input_line` variable of type string is declared on the following line and initialized to string.Empty. The `while` loop on line 19 processes each line of the text file. It first formulates a line number string (`line_number_string`) and checks its existence within the hashtable via the Contains() method. If it's not in the hashtable, the `input_line` is added using the `line_number_string` as its key.

The `for` statement on line 41 steps through each element of the hashtable using its indexer and writing the retrieved value to the console. Lines 46 through 49 access individual elements of the hashtable via the indexer.

The text file used for this example is named Book.txt. It contains the complete text of Cicero's Tusculan Disputations, by Marcus Tullius Cicero. It was downloaded from the Project Gutenberg website. (www.gutenberg.net)

Figure 9-5 shows the results of running this program. Note that figure 9-5 only shows the last few lines of the output of 18517 lines of text printed to the console.

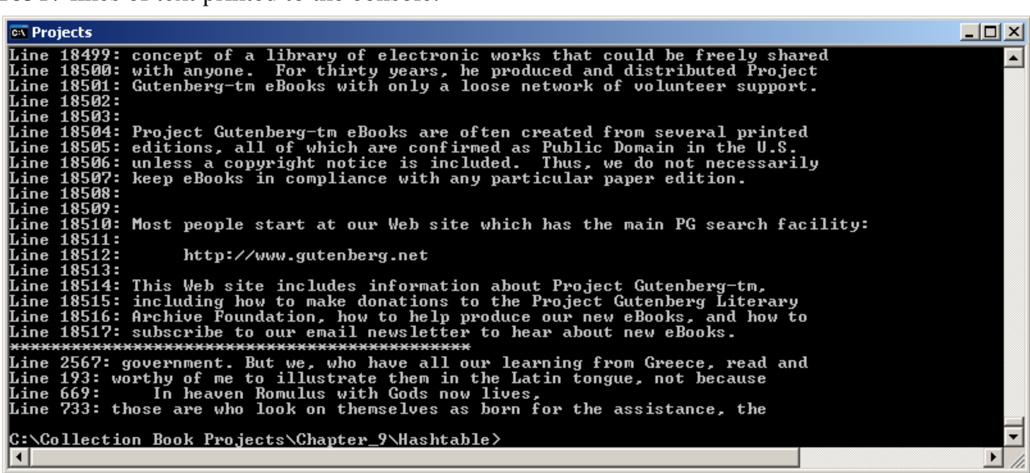


Figure 9-5: Results of Running Example 9.4

Quick Review

The Hashtable is a non-generic collection class that stores its item as DictionaryEntry objects. The Hashtable class resolves collisions via *double hashing*, which is a process by which a key is rehashed using a modified hashing function until the collision has been resolved by the generation of a unique bucket location.

Dictionary<TKey, TValue> Class

The Dictionary<TKey, TValue> class is the strongly-typed, generic version of the non-generic Hashtable class. The Dictionary<TKey, TValue> class also differs from the Hashtable class in the way it handles collisions. It uses chaining. Like the HomeGrownHashtable presented earlier, values whose keys hash to the same bucket are stored in a list, however, unlike the HomeGrownHashtable example, the Dictionary<TKey, TValue> class uses a different chain management algorithm which I'm positive is much more efficient than the approach I used.

Another huge difference between the Dictionary<TKey, TValue> class and the Hashtable class is the large number of extension methods provided by the System.Linq.Enumerable class that can be used on the Dictionary.

Figure 9-6 offers a UML class diagram showing the inheritance hierarchy of the Dictionary<TKey, TValue> class.

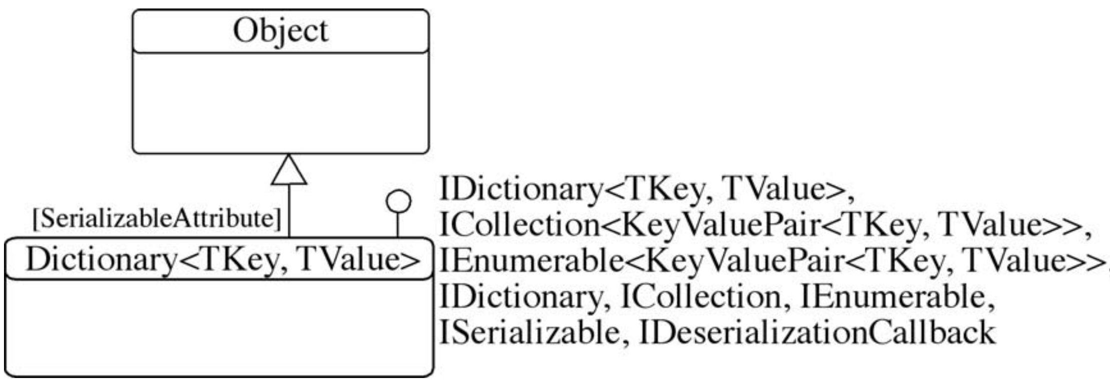


Figure 9-6: Dictionary<TKey, TValue> Class Inheritance Hierarchy

Referring to figure 9-6 — the Dictionary<TKey, TValue> class implements the IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, IEnumerable, ISerializable, and IDeserializationCallback interfaces. Each of these interfaces is discussed in greater detail below.

Functionality Provided by the IEnumerable and IEnumerable<KeyValuePair<TKey, TValue>> Interfaces

The IEnumerable and IEnumerable<KeyValuePair<TKey, TValue>> interfaces allow the items within a Dictionary<TKey, TValue> class to be iterated over with a `foreach` statement. Note that each element within a Dictionary<TKey, TValue> collection is a KeyValuePair<TKey, TValue> object. Assuming there was a reference to a Dictionary<string, int> object named names_and_ages, you could step through each element of the collection using a `foreach` statement similar to the following code snippet:

```
foreach(KeyValuePair<string, int> entry in names_and_ages){
  Console.WriteLine("{0} is {1} years old!", entry.Key, entry.Value);
}
```

Functionality Provided by the ICollection and ICollection<KeyValuePair<TKey, TValue>> Interfaces

The ICollection and ICollection<KeyValuePair<TKey, TValue>> interfaces tag the Dictionary<TKey, TValue> class as a collection type. The ICollection interface declares object synchronization properties IsSynchronized and SyncRoot, while the ICollection<KeyValuePair<TKey, TValue>> interface declares the Add(), Remove(), and Con-

tains() methods and the Count property. These interfaces also declare the GetEnumerator() methods required to iterate over the collection with a `foreach` statement.

Functionality Provided by the IDictionary and IDictionary<KeyValuePair<TKey, TValue>> Interfaces

The IDictionary and IDictionary<KeyValuePair<TKey, TValue>> interfaces provide the non-generic and generic versions of Keys and Values properties, the indexer, and the ContainsKey() and the TryGetValue() methods.

Functionality Provided by the ISerializable and IDeserializationCallback Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the Dictionary<TKey, TValue> collection requires custom serialization code over and beyond what the Serializable attribute alone provides.

Dictionary<TKey, TValue> Example

Example 9.5 presents a short program demonstrating the use of the Dictionary<TKey, TValue> collection.

9.5 DictionaryDemo.cs

```
1        using System;
2        using System.Collections.Generic;
3        using System.Linq;
4
5        public class DictionaryDemo {
6
7           public static void Main(){
8             Dictionary<string, int> names_and_ages = new Dictionary<string, int>();
9             names_and_ages.Add("Rick", 49);
10            names_and_ages.Add("Kyle", 23);
11            names_and_ages.Add("Sport", 39);
12            names_and_ages.Add("Coralie", 39);
13            names_and_ages.Add("Tati", 21);
14            names_and_ages.Add("Schmoogle", 7);
15
16            foreach(KeyValuePair<string, int> entry in names_and_ages){
17              Console.WriteLine("{0} is {1} years old!", entry.Key, entry.Value);
18            }
19
20            Console.WriteLine("The average age is {0:F4}", names_and_ages.Values.Average());
21
22          } //  end Main() method
23       } // end DictionaryDemo class
```

Referring to example 9.5 — a Dictionary<string, int> reference named `names_and_ages` is declared and created on line 8. In this case, the keys will be strings and the values will be integers. Lines 9 through 14 add several entries into the dictionary. The `foreach` statement on line 16 steps through each KeyValuePair entry in the dictionary and prints the key and value to the console. Line 20 extracts the values from the dictionary via the Values property and calls the extension method Average() to calculate the average age. Figure 9-7 shows the results of running this program.

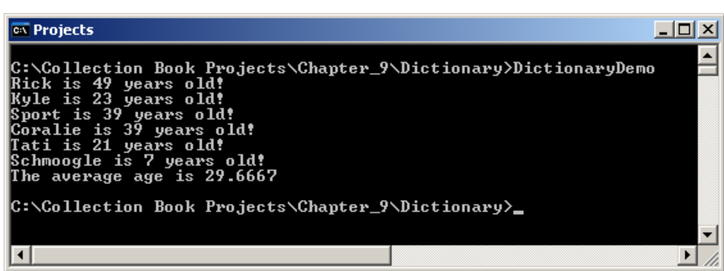


Figure 9-7: Results of Running Example 9.5

Quick Review

The Dictionary<TKey, TValue> collection is a strongly-typed version of the Hashtable class that uses chaining instead of double hashing to resolve collisions. The Dictionary<TKey, TValue> collection stores its items as KeyValuePair objects. (See System.Collections.Generic.KeyValuePair<TKey, TValue> structure.)

Summary

A hashtable is an array that employs a special operation called a *hash function* to calculate an object's location within the array. The object being inserted into the table is referred to as the *value*. A hash function is applied to the value's associated *key* which results in an integer value that lies somewhere between the first array element (0) and the last array element (n-1).

The HomeGrownHashtable class implements a chained hashtable where each hashtable *bucket* points to a List<KeyValuePair> object into which KeyValuePair objects are inserted. The hashtable's load limit determines when the hashtable should be grown to accommodate additional elements. The load limit is calculated by multiplying the table size by the load factor.

Hashtable *collisions* can occur when two different keys hash to the same hash value. The chained hash table resolves collisions by allowing collisions to occur and storing the KeyValuePair objects in a list at that location which must then be searched to find the key/value pair of interest.

The Hashtable is a non-generic collection class that stores its item as DictionaryEntry objects. The Hashtable class resolves collisions via *double hashing*, which is a process by which a key is rehashed using a modified hashing function until the collision has been resolved by the generation of a unique bucket location.

The Dictionary<TKey, TValue> collection is a strongly-typed version of the Hashtable class that uses chaining instead of double hashing to resolve collisions. The Dictionary<TKey, TValue> collection stores its items as KeyValuePair objects. (See System.Collections.Generic.KeyValuePair<TKey, TValue> structure.)

References

Donald E. Knuth. The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Third Edition. Addison-Wesley, Reading, Massachusetts. 1997. ISBN: 0-201-89683-4.

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Arash Partow. The General Purpose Hash Function Algorithms Library (GeneralHashFunctionLibrary.java). [http://www.partow.net/programming/hashfunctions/index.html]

Project Gutenberg [http://www.gutenberg.net]

Notes

Chapter 10


Yashica Mat 124G

Amsterdam Bridge

Coding For Collections

Learning Objectives

- *Enable user-defined data types to perform correctly in collections*
- *Define the term "natural ordering"*
- *State the difference between natural ordering and custom ordering*
- *Create classes and structures that can be used in equality comparisons*
- *Override System.Object.Equals() and System.Object.GetHashCode() methods*
- *Implement the IComparable and IComparable<T> interfaces to specify natural ordering*
- *Implement the IComparer and IComparer<T> interfaces to create a custom comparer*
- *Implement the IEquatable interface to allow objects to be used as keys*
- *Define the term "immutable" object*

Introduction

When creating user-defined data types you must stop for a moment to consider how they will be used in your program. If you intend to use them in collections then you must enable them to be used in equality and comparison operations. For example, if you intend to sort user-defined objects using the Array.Sort() method, then you must provide the ability for one object to be compared with another for the sort operation to work correctly. If you intend to use user-defined data types as keys in hashtables, dictionaries, or other keyed collections, then you'll need to know how to get your objects to behave correctly as keys. These topics are the focus of this chapter.

I'll start by showing you how to override the Object.Equals() and Object.GetHashCode() methods. I'll then explain why and how to overload the == and != operators.

Next I'll talk about comparison operations and show you how to specify *natural ordering* by implementing the IComparable and IComparable<T> interfaces. Following this I'll show you how to create individual comparer objects that are used to specify *custom ordering* by implementing the IComparer and IComparer<T> interfaces.

I wrap up the chapter by showing you how to create objects that can be used as keys in hashtables, dictionaries, and other keyed collections. This includes a discussion of *object immutability.*

Upon completing this chapter you'll have a thorough understanding of how to create user-defined types that behave well when used in collections. Now, let's get going!

Coding for Equality Operations

Objects of a particular type, when used in non-keyed collections like arrays and lists, must be able to be used in equality comparison operations. This section discusses the differences between *reference equality*, *value equality*, and *bitwise equality*, and shows you how to override the Object.Equals() and Object.GetHashCode() methods. Following this I'll show you how to overload the == and != operators.

Reference Equality vs. Value Equality

Normally, when you compare two reference objects for equality like this...

```
o1 == o2
```

...you are comparing their addresses. In other words, if o1 and o2 refer to the same location in memory then they must be equal because they refer to the same object. However, it's not always desirable to use an object's address as a basis for equality. Take strings for example. Two strings of equal value may be different objects as the following code snippet suggests:

```
String s1 = "Hello";
String s2 = "Hello";
```

The expression (`s1 == s2`) will yield true just as `s1.Equals(s2)` will yield true. This is because the Equals() method has been overridden and the == operator has been overloaded to perform a *value* or string content comparison, which is what you'd expect when comparing two strings.

For structures, the default behavior of the Object.Equals() method and the == operator is *bitwise equality*. For the most part, bitwise equality means the same thing as value equality, especially in the case of simple value types. (i.e., structures like Int32) If, however, the binary representation of the value type is complex, like the Decimal structure, then the Object.Equals() method is overridden and the == operator is overloaded to yield the expected value comparison behavior. For example, given two integer variables:

```
int i = 1;
int j = 2;
```

The expression (`i == j`) compares the value of i, which is 1, against the value of j, which is 2. In either case you can substitute the == operator with the Equals() method like so:

```
i.Equals(j);
```

Overriding Object.Equals() and Object.GetHashCode()

If the default behavior of the Object.Equals() method is insufficient for your user-defined data types, you'll need to override it and provide a custom implementation. Both the Object.Equals() and Object.GetHashCode() methods must be overridden together to ensure correct behavior. The following sections present the rules that should be followed when overriding these methods.

Rules For Overriding The Object.Equals() Method

When overriding the Object.Equals() method, you must ensure that it subscribes to the expected behavior as specified in the .NET Framework documentation. Table 10.1 lists the required behavior of an overridden Object.Equals() method. (**Note:** The overloaded **==** operator must work the same way!)

Should be...	Rule	Comment
Reflexive	x.Equals(x) returns true	Exception: floating-point types
Symmetric	x.Equals(y) returns the same as y.Equals(x)	
Transitive	(x.Equals(y) && y.Equals(z)) returns true if and only if x.Equals(z) returns true	
Consistent	Successive calls to x.Equals(y) return the same value as long as the objects referenced by x and y remain unchanged.	
	x.Equals(null) returns false	Or a null reference
	x.Equals(y) returns true if both x and y are NaN	NaN means Not a Number
	Calls to Object.Equals() must not throw exceptions.	No exceptions!
	Override the Object.GetHashCode() method.	If you override the Object.Equals() method.

Table 10-1: Rules for Overriding Object.Equals() method

Rules For Overriding The Object.GetHashCode() Method

When you override the Object.Equals() method you should also override the Object.GetHashCode() method to ensure proper object behavior. This section presents two approaches to implementing a suitable GetHashCode() method. Now, don't be alarmed when I reference two very good Java books. The techniques used to create a suitable hashcode algorithm apply equally to C# as well as Java.

The GetHashCode() method returns an integer which is referred to as the object's *hash value*. The default implementation of GetHashCode() found in the Object class will, in most cases, return a unique hash value for each distinct object even if they are logically equivalent. In most cases this default behavior is acceptable, however, if you intend to use a class of objects as keys to hashtables or other hash-based data structures, then you must override the GetHashCode() method and obey the general contract as specified in the .NET Framework API documentation. The general contract for the GetHashCode() is given in Table 10-2.

Check	Criterion
	The GetHashCode() method must consistently return the same integer when invoked on the same object more than once during an execution of a C# or .NET application, provided no information used in Equals() comparisons on the object is modified. This integer need not remain constant from one execution of an application to another execution of the same application.

Table 10-2: The GetHashCode() General Contract

Check	Criterion
	The GetHashCode() method must produce the same results when called on two objects if they are equal according to the Equals() method.
	The GetHashCode() method is not required to return distinct integer results for logically unequal objects, however, failure to do so may result in degraded hash table performance.

Table 10-2: The GetHashCode() General Contract

As you can see from Table 10-2 there is a close relationship between the Object.Equals() and Object.GetHashCode() methods. It is recommended that any fields used in the Equals() method comparison be used to calculate an object's hash code. Remember, the primary goal when implementing a GetHashCode() method is to have it return the same value consistently for logically equal objects. It would also be nice if the GetHashCode() method returned distinct hash code values for logically unequal objects, but according to the general contract this is not a strict requirement.

Before actually implementing a GetHashCode() method, I want to provide you with two hash code generation algorithms. These algorithms come from two excellent Java references. (Yes, I meant to say Java.) I have changed the text to reflect the .NET method names Object.Equals() and Object.GetHashCode() respectively, and have converted Java operations into compatible C# .NET operations.

Bloch's Hash Code Generation Algorithm

Joshua Bloch, in his book *Effective Java™ Programming Language Guide*, provides the following algorithm for calculating a hash code:

1. Start by storing a constant, nonzero value in an `int` variable called `result`. (Josh used the value 17)
2. For each significant field *f* in your object (each field involved in the Equals() comparison) do the following:
    a. Compute an `int` hash code `c` for the field:
        i. If the field is boolean (bool) compute: `(f?0:1)`
        ii. If the field is a byte, char, short, or int, compute: `(int)f`
        iii. If the field is a long compute: `(unsigned)(f^(f >> 32))`
        iv. If the field is a float compute: `Convert.ToInt32(f)`
        v. If the field is a double compute: `Convert.ToInt64(f)`, and then hash the resulting long according to step 2.a.iii.
        vi. If the field is an object reference and this class's Equals() method compares the field by recursively invoking Equals(), recursively invoke GetHashCode() on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke GetHashCode() on the canonical representation. If the value of the field is null, return 0.
        vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values in step 2.b
    b. Combine the hash code `c` computed in step a into `result` as follows:
        `result = 37*result + c;`
3. Return `result`.
4. If equal object instances do not have equal hash codes fix the problem!

Ashmore's Hash Code Generation Algorithm

Derek Ashmore, in his book *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications*, recommends the following simplified hash code algorithm:

1. Concatenate the required fields (those involved in the Equals() comparison) into a string.
2. Call the GetHashCode() method on that string.
3. Return the resulting hash code value.

An Example: The Person Class

I'll use a class named Person to demonstrate how to override the Object.Equals() and Object.GetHashCode() methods. Example 10.1 lists the code for the Person class.

10.1 Person.cs (Overridden Equals() and GetHashCode() Methods)

```
using System;

public class Person {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;

  public Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
    get { return _firstName; }
    set { _firstName = value; }
  }

  public String MiddleName {
    get { return _middleName; }
    set { _middleName = value; }
  }

  public String LastName {
    get { return _lastName; }
    set { _lastName = value; }
  }

  public Sex Gender {
    get { return _gender; }
    set { _gender = value; }
  }

  public DateTime Birthday {
    get { return _birthday; }
    set { _birthday = value; }
  }

  public Guid DNA {
    get { return _dna; }
```

```
75          set {  _dna = value; }
76        }
77
78        public int Age {
79          get {
80            int years = DateTime.Now.Year - _birthday.Year;
81            int adjustment = 0;
82            if(DateTime.Now.Month < _birthday.Month){
83              adjustment = 1;
84          } else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
85                   adjustment = 1;
86                 }
87            return years - adjustment;
88          }
89        }
90
91        public String FullName {
92          get {  return FirstName + " " + MiddleName + " " + LastName; }
93        }
94
95        public String FullNameAndAge {
96          get {  return FullName + " " + Age; }
97        }
98
99        public override String ToString(){
100         return (FullName + "  " + Gender + "  " + Age + " " + DNA);
101       }
102
103       public override bool Equals(object o){
104         if(o == null) return false;
105         if(typeof(Person) != o.GetType()) return false;
106         return this.ToString().Equals(o.ToString());
107       }
108
109       public override int GetHashCode(){
110         return this.ToString().GetHashCode();
111       }
112
113     } // end Person class
```

Referring to example 10.1 — the Person class defines the usual fields you'd expect for a data type of this nature. I've also added a field called _dna of type Guid (Globally Unique Identifier). (I know, I'm being cheeky here calling the field _dna. In real life, the name of this field might be _id which would map to the primary key column of a relational database table where state values of person objects are persisted.) I've added the _dna field with its corresponding Guid type to make it easier to make Person objects unique.

The overridden Object.ToString() method is defined on line 99. It returns a concatenation of the FullName, Gender, Age, and DNA properties. (The Age property is an example of a calculated read-only property.) The overridden Object.Equals() method starts on line 103. It relies on the ToString() method to compare different person objects for value equality. The GetHashCode() method simply calls the GetHashCode() method on the string generated by the Person object's ToString() method.

Example 10.2 gives the code for a short application that creates a few Person objects and tests the Object.Equals() method, validating its conformance to the rules laid out in table 10-1.

10.2 MainApp.cs (Demonstrating Overridden Equals() & GetHashCode() Methods)

```
1       using System;
2
3       public class MainApp {
4         public static void Main(){
5           Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
6                       new DateTime(1961, 2, 3), Guid.NewGuid());
7           Console.WriteLine("p1.Equals(p1) : {0}", p1.Equals(p1));
8           Console.WriteLine("p1.Equals(string) : {0}", p1.Equals("Hello!"));
9           Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
10                      new DateTime(1972, 1, 1), Guid.NewGuid());
11          Console.WriteLine("p1.Equals(p2) : {0}", p1.Equals(p2));
12          Console.WriteLine("p2.Equals(p1) : {0}", p2.Equals(p1));
13          Console.WriteLine("p1.GetHashCode() = {0}", p1.GetHashCode());
14          Console.WriteLine("p2.GetHashCode() = {0}", p2.GetHashCode());
15        }
16      }
```

Referring to example 10.2 — On line 5 a Person reference named p1 is created and initialized. The Object.Equals() method is then called using the reference p1 as an argument. This of course should return true. Next, p1 is compared with a string object, which should return false. On line 9 a second Person reference named p2 is declared and initialized and it's compared with p1. Both tests should return false. Following this, the GetHashCode()

method is called on each reference. The values returned by these last two method calls will yield different values when you run this program on your computer. Figure 10-1 shows the results of running this program.

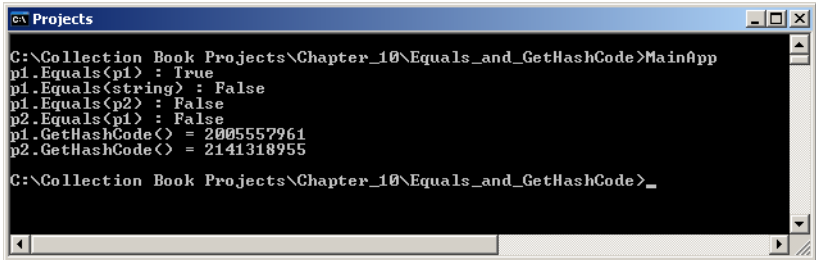


Figure 10-1: Results of Running Example 10.2

Overloading the == and != Operators

Although not strictly required to be overloaded for the purposes of collections, the == and != operators can be overloaded with little effort because they can simply use the overridden Object.Equals() method in their implementation. (Low hanging fruit!) Example 10.3 gives the modified Person class with the overloaded == and != operators.

10.3 Person.cs (Overloaded == and != Operators)

```
using System;

public class Person {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
```



```
51          get {  return _firstName; }
52          set {  _firstName = value; }
53        }
54
55        public String MiddleName {
56          get {  return _middleName; }
57          set {  _middleName = value; }
58        }
59
60        public String LastName {
61          get {  return _lastName; }
62          set {  _lastName = value; }
63        }
64
65        public Sex Gender {
66          get {  return _gender; }
67          set {  _gender = value; }
68        }
69
70        public DateTime Birthday {
71          get {  return _birthday; }
72          set {  _birthday = value; }
73        }
74
75        public Guid DNA {
76          get {  return _dna; }
77          set {  _dna = value; }
78        }
79
80        public int Age {
81          get {
82            int years = DateTime.Now.Year - _birthday.Year;
83            int adjustment = 0;
84            if(DateTime.Now.Month < _birthday.Month){
85              adjustment = 1;
86            } else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
87                    adjustment = 1;
88                  }
89            return years - adjustment;
90          }
91        }
92
93        public String FullName {
94          get {  return FirstName + " " + MiddleName + " " + LastName; }
95        }
96
97        public String FullNameAndAge {
98          get {  return FullName + " " + Age; }
99        }
100
101       public override String ToString(){
102         return (FullName + "  " + Gender + "  " + Age + " " + DNA);
103       }
104
105       public override bool Equals(object o){
106         if(o == null) return false;
107         if(typeof(Person) != o.GetType()) return false;
108         return this.ToString().Equals(o.ToString());
109       }
110
111       public override int GetHashCode(){
112         return this.ToString().GetHashCode();
113       }
114
115       public static bool operator ==(Person lhs, Person rhs){
116         return lhs.Equals(rhs);
117       }
118
119       public static bool operator !=(Person lhs, Person rhs){
120         return !(lhs.Equals(rhs));
121       }
122
123     } // end Person class
```

Referring to example 10.3 — the == operator is overloaded on line 115. Note that it's a static method and that it defines two method parameters of type Person named lhs (left hand side) and rhs (right hand side). It simply calls the overridden Object.Equals() method to make the equality check. It can do this because the rules for overloading the == operator are the same as the rules for overriding the Object.Equals() method, so each must exhibit the same behavior.

The != operator is defined on line 119. It too relies on the overridden Object.Equals() method in its implementation. Note that it simply negates the result of comparing the lhs with the rhs with the Equals() method.

Example 10.4 demonstrates the use of the overloaded == and != operators.

10.4 MainApp.cs (Demonstrating Overloaded == and != Operators)

```
1       using System;
2
3       public class MainApp {
4         public static void Main(){
5           Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
6                       new DateTime(1961, 2, 3), Guid.NewGuid());
7           Console.WriteLine("p1.Equals(p1) : {0}", p1.Equals(p1));
8           Console.WriteLine("p1.Equals(string) : {0}", p1.Equals("Hello!"));
9           Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
10                      new DateTime(1972, 1, 1), Guid.NewGuid());
11          Console.WriteLine("p1.Equals(p2) : {0}", p1.Equals(p2));
12          Console.WriteLine("p2.Equals(p1) : {0}", p2.Equals(p1));
13          Console.WriteLine("p1.GetHashCode() = {0}", p1.GetHashCode());
14          Console.WriteLine("p2.GetHashCode() = {0}", p2.GetHashCode());
15          Console.WriteLine("p1 == p1 : {0}", p1 == p1);
16          Console.WriteLine("p1 == p2 : {0}", p1 == p2);
17          Console.WriteLine("p1 != p1 : {0}", p1 != p1);
18          Console.WriteLine("p1 != p2 : {0}", p1 != p2);
19        }
20      }
```

Referring to example 10.4 — the tests of the == and != operators have been added to the previous MainApp example. On line 15 the reference p1 is compared with itself using the == operator and again on line 17 using the != operator. These comparisons result in the compiler warnings shown in figure 10-2. You can safely ignore them here for the sake of testing. Figure 10-3 shows the results of running this program.

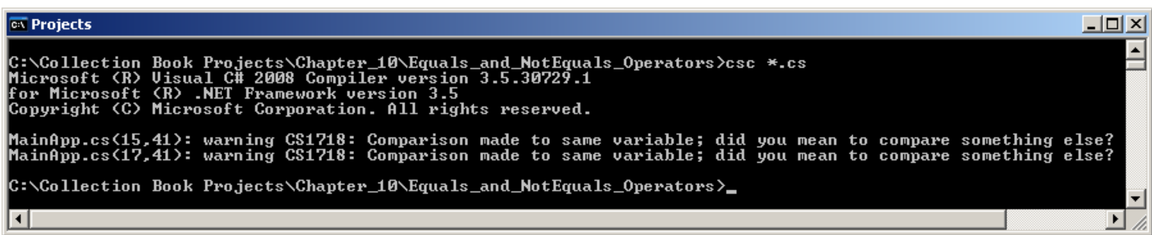


Figure 10-2: Compiler Warning Generated when Compiling Examples 10.3 and 10.4

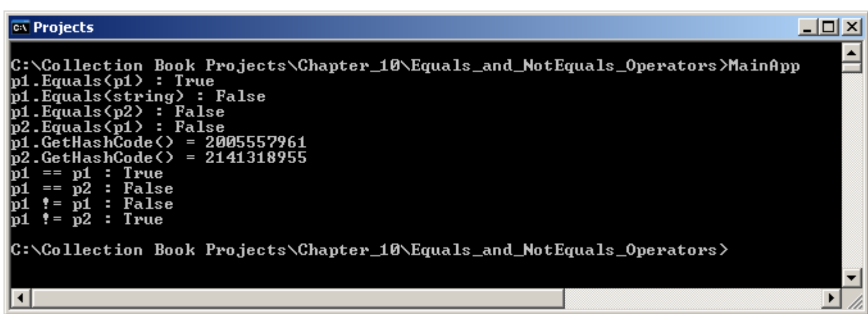


Figure 10-3: Results of Running Example 10.4

Quick Review

The first step in getting your user-defined types to behave well in collections is to override the Object.Equals() and Object.GetHashCode() methods. Make sure you adhere to the Object.Equals() method behavior rules. You can optionally overload the == and != methods as their behavior can be easily implemented in terms of the Object.Equals() method.

The overridden Object.GetHashCode() method can be easily implemented by calling the GetHashCode() method on the string returned by the object's overridden ToString() method.

Coding for Comparison Operations

If you intend to insert user-defined objects into a collection and sort them you'll need to define how, exactly, one object is to be compared with another in terms of being *less than*, *equal to*, or *greater than* another object. You do this by implementing either the *IComparable* or the *IComparable<T>* interfaces, or both if you plan to use user-defined objects in both non-generic and generic collections. In this section I explain the concept of natural ordering and show you how to implement each of these interfaces.

Natural Ordering

When you implement the IComparable and IComparable<T> interfaces in a class or structure you are specifying what is referred to as a *natural ordering* for that particular type. It's called natural ordering because you have instructed the type how to behave when compared with other objects of the same (or different) type.

Take integers for example. If you examine the .NET documentation for the Int32 structure you'll see that it implements both the IComparable and IComparable<T> (as IComparable<int>) interfaces. This allows integers to be compared with other integers when sorted with the Sort() method defined by the Array class and other collections that allow elements to be sorted.

IComparable and IComparable<T> Interfaces

The IComparable and IComparable<T> interfaces each declare one method named CompareTo(object other) that returns an integer, the value of which must reflect the results of the comparison as listed in the rules shown in table 10-3.

Return Value	Returned When...
Less than Zero (-1)	This object is less than the *other* parameter
Zero (0)	This object is equal to the *other* parameter
Greater than Zero (1)	This object is greater than the *other* parameter, or, the *other* parameter is null

Table 10-3: Rules For Implementing IComparable.CompareTo() Method

Referring to table 10-3 — as the rules state, if the object (represented by the `this` reference) is less than the `other` parameter, the CompareTo() method returns some value less than 0. (The value -1 is fine.) If both objects being compared are equal it returns 0, and if the `other` object is greater or *null* it returns a positive number. (1 is fine.) Example 10.5 shows how the IComparable and IComparable<T> interfaces can be implemented in the Person class.

10.5 Person.cs (Implementing IComparable and IComparable<T> Interfaces)

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

```



```
      public Person(String firstName, String middleName, String lastName,
                    Sex gender, DateTime birthday, Guid dna){
         FirstName = firstName;
         MiddleName = middleName;
         LastName = lastName;
         Gender = gender;
         Birthday = birthday;
         DNA = dna;
      }

      public Person(String firstName, String middleName, String lastName,
                    Sex gender, DateTime birthday){
         FirstName = firstName;
         MiddleName = middleName;
         LastName = lastName;
         Gender = gender;
         Birthday = birthday;
         DNA = Guid.NewGuid();
      }

      public Person(Person p){
         FirstName = p.FirstName;
         MiddleName = p.MiddleName;
         LastName = p.LastName;
         Gender = p.Gender;
         Birthday = p.Birthday;
         DNA = p.DNA;
      }

      // public properties
      public String FirstName {
        get { return _firstName; }
        set { _firstName = value; }
      }

      public String MiddleName {
        get { return _middleName; }
        set { _middleName = value; }
      }

      public String LastName {
        get { return _lastName; }
        set { _lastName = value; }
      }

      public Sex Gender {
        get { return _gender; }
        set { _gender = value; }
      }

      public DateTime Birthday {
        get { return _birthday; }
        set { _birthday = value; }
      }

      public Guid DNA {
        get { return _dna; }
        set { _dna = value; }
      }

      public int Age {
        get {
          int years = DateTime.Now.Year - _birthday.Year;
          int adjustment = 0;
          if(DateTime.Now.Month < _birthday.Month){
             adjustment = 1;
          }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
                 adjustment = 1;
          }
          return years - adjustment;
        }
      }

      public String FullName {
        get { return FirstName + " " + MiddleName + " " + LastName; }
      }

      public String FullNameAndAge {
        get { return FullName + " " + Age; }
      }

```



```
101       protected String SortableName {
102         get {  return LastName + FirstName + MiddleName; }
103       }
104
105       public override String ToString(){
106         return (FullName + "  " + Gender + "  " + Age + " " + DNA);
107       }
108
109       public override bool Equals(object o){
110         if(o == null) return false;
111         if(typeof(Person) != o.GetType()) return false;
112         return this.ToString().Equals(o.ToString());
113       }
114
115       public override int GetHashCode(){
116         return this.ToString().GetHashCode();
117       }
118
119       public static bool operator ==(Person lhs, Person rhs){
120         return lhs.Equals(rhs);
121       }
122
123       public static bool operator !=(Person lhs, Person rhs){
124         return !(lhs.Equals(rhs));
125       }
126
127       public int CompareTo(object obj){
128         if((obj == null) || (typeof(Person) != obj.GetType()))  {
129           throw new ArgumentException("Object is not a Person!");
130         }
131         return this.SortableName.CompareTo(((Person)obj).SortableName);
132       }
133
134       public int CompareTo(Person p){
135         if(p == null){
136           throw new ArgumentException("Cannot compare null objects!");
137         }
138         return this.SortableName.CompareTo(p.SortableName);
139       }
140
141     } // end Person class
```

Referring to example 10.5 — on line 3 the IComparable and IComparable<T> interfaces are listed as being implemented by the Person class. Note how the IComparable<T> interface actually reads IComparable<*Person*>. The non-generic CompareTo() method begins on line 127. This version of the method corresponds with the IComparable interface. It takes an object argument and must test it to see if it's the proper type. If it's not, or it's *null*, it throws an ArgumentException.

The CompareTo() method on line 134 corresponds to the IComparable<Person> interface. Note that since the type of parameter has been specified, it's no longer necessary to explicitly test the incoming object for type conformance, as this is handled by the compiler.

Also important to note here is how I've defined natural ordering for Person objects. I've chosen to order Person object's by last names, first names, and middle names. To help in this effort I have added another property to the Person class named SortableName which concatenates the name fields together for proper sorting.

Example 10.6 demonstrates how an array of Person objects can now be sorted by name.

10.6 MainApp.cs (Sorting and Array of Person Objects with Natural Ordering)

```
using System;

public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
                 new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
                 new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
                 new DateTime(1959, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
                 new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
                 new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
                 new DateTime(1986, 10, 15), Guid.NewGuid());

    Person[] people_array = new Person[6];
    people_array[0] = p1;
    people_array[1] = p2;
```



```
21          people_array[ 2] = p3;
22          people_array[ 3] = p4;
23          people_array[ 4] = p5;
24          people_array[ 5] = p6;
25
26          Console.WriteLine("-------- Before Sorting ------------");
27
28          foreach(Person p in people_array){
29            Console.WriteLine(p.LastName + "," + p.FirstName);
30          }
31
32          Array.Sort(people_array);
33
34          Console.WriteLine("-------- After Sorting ------------");
35
36          foreach(Person p in people_array){
37            Console.WriteLine(p.LastName + "," + p.FirstName);
38          }
39        }
40      }
```

Referring to example 10.6 — the six Person objects created on lines 5 through 16 are used to initialize the six elements of the people_array on lines 19 through 24. The `foreach` statement on line 28 prints out the contents of the array to the console before sorting. The `foreach` statement on line 36 does the same after the array has been sorted. The Array.Sort() method called on line 32 expects the elements in the array passed to it as an argument to implement IComparable. If one or more elements in the array fail to implement IComparable, the Sort() method will throw an InvalidOperationException. Figure 10-4 shows the results of running this program.

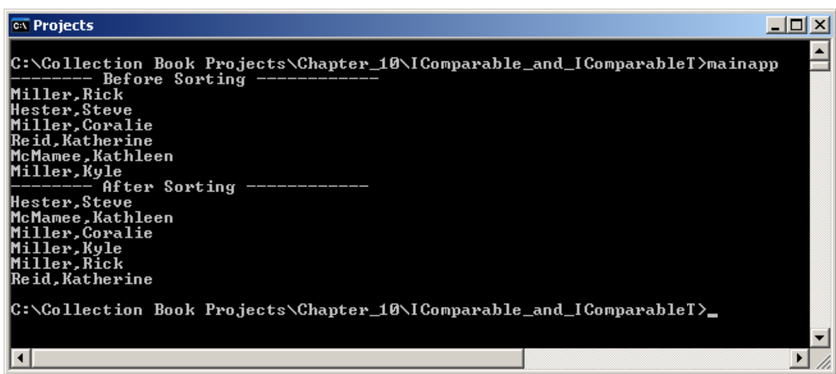


Figure 10-4: Results of Running Example 10.6

Custom Ordering: Creating Separate Comparer Objects

As you learned in the preceding section, to specify a natural ordering for your user-defined types you must implement the IComparable and IComparable<T> interfaces. If you want to order objects in a different way, you can create custom comparers by implementing the IComparer and IComparer<T> interfaces.

IComparer and IComparer<T> Interfaces

The IComparer and IComparer<T> interfaces both declare one method named Compare(). In the case of IComparer the method signature is *int Compare(object x, object y)* and for IComparer<T> it's *int Compare(T x, T y)*. The rules for implementing the Compare() methods are the same ones used to implement the CompareTo() methods discussed in the previous section.

These methods are easy to implement. In most cases, custom ordering boils down to one particular field within the user-defined type. For example, if you want to provide a custom ordering of Person objects by age, you would simply be comparing two integers: one person object's age against another's. And since all the built-in .NET types already implement the IComparable and IComparable<T> interfaces, you can implement the Compare() method in terms of each object's CompareTo() method.

An Example: PersonAgeComparer

Example 10.7 gives the code for a class named PersonAgeComparer. The PersonAgeComparer class implements both the IComparer and IComparer<T> interfaces.

10.7 PersonAgeComparer.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4
5   public class PersonAgeComparer : IComparer, IComparer<Person> {
6
7     public int Compare(object x, object y){
8       if((x == null) || (y == null) || (typeof(Person) != x.GetType())
9                    || (typeof(Person) != y.GetType())){
10        throw new ArgumentException("Both objects must be of type Person!");
11      }
12
13      return ((Person)x).Age.CompareTo(((Person)y).Age);
14    }
15
16    public int Compare(Person x, Person y){
17     if((x == null) || (y == null)){
18        throw new ArgumentException("Both objects must be of type Person!");
19      }
20
21      return x.Age.CompareTo(y.Age);
22    }
23
24  }
```

Referring to example 10.7 — the non-generic Compare() method starts on line 7. The `if` statement on line 8 checks to ensure incoming arguments are valid Person objects. If the arguments fail this test the method throws an ArgumentException. Line 13 contains the meat of the method: It casts each parameter to type Person and calls the CompareTo() method via the x parameter passing the y parameter as an argument. Done!

The generic version of the Compare() method on line 16 safely skips the type testing part of the `if` statement since the method parameters already specify the type. If the arguments are *null* it throws an ArgumentException, otherwise, the comparison of the x parameter with the y parameter proceeds without the casting as was necessary in the non-generic version of the Compare() method.

Example 10.8 demonstrates the use of the PersonAgeComparer class.

10.8 MainApp.cs (Demonstrating Custom Ordering with PersonAgeComparer)

```
using System;


public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
                new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
                new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
                new DateTime(1974, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
                new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
                new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
                new DateTime(1986, 10, 15), Guid.NewGuid());

    Person[] people_array = new Person[6];
    people_array[0] = p1;
    people_array[1] = p2;
    people_array[2] = p3;
    people_array[3] = p4;
    people_array[4] = p5;
    people_array[5] = p6;

    Console.WriteLine("-------- Before Sorting ------------");

    foreach(Person p in people_array){
      Console.WriteLine(p.FullNameAndAge);
    }

    Array.Sort(people_array, new PersonAgeComparer());

    Console.WriteLine("-------- After Sorting ------------");
```



```
36
37          foreach(Person p in people_array){
38            Console.WriteLine(p.FullNameAndAge);
39          }
40        }
41      }
```

Referring to example 10.8 — note on line 33 that a PersonAgeComparer object is passed as the second argument to the Array.Sort() method. If a custom comparer object is supplied to the Array.Sort() method, as is done here, it orders the elements in the array according to the custom comparer. The result in this case is that the elements are sorted by age vs. last, first, and middle names. Figure 10-5 shows the results of running this program.

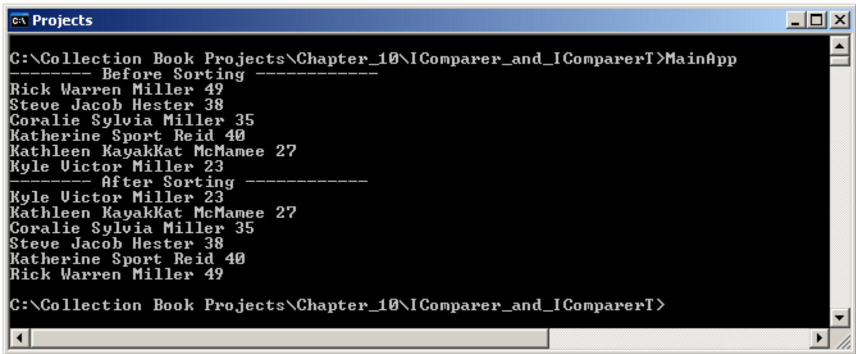


Figure 10-5: Results of Running Example 10.8

Quick Review

Implement both the IComparable and IComparable<T> interfaces to specify a natural ordering for user-defined types. Implement the IComparer and IComparer<T> interfaces to create a custom comparer. Custom comparers are used to specify a custom ordering. You can create as many custom comparers as required.

It's a good idea to always implement both the generic and non-generic versions of these interfaces. Doing so ensures your user-defined types will be sortable in generic and non-generic collections.

Using Objects as Keys

In keyed collections, objects are inserted into the collection in key/value pairs. Object's used as keys must obey certain rules. This section explains those rules and demonstrates how to create a type suitable for the creation of key objects.

Rules For Objects Used As Keys

Objects inserted into keyed collections are located within those collections via an operation performed upon their associated key. In chapter 9 you learned about the Hashtable and Dictionary<T Key, T Value> collections. In these collections, the value's location with the hash table is determined by applying a hash function to the key. Before an object can be used as a key it must adhere to a few rules as listed in table 10-4.

Rule	Comment
Key must be immutable	Objects used as keys must not change value while they are being used as keys. Object's whose state value cannot be changed after they are created are called immutable objects. Strings are immutable objects, which is why they can be safely used as keys.

Table 10-4: Rules For Creating Key Classes

Rule	Comment
Implement the IEquatable<T> interface	The IEquatable<T> interface is used by generic collections to test keys for equality. It defines one method named Equals().
Override the Object.Equals() method	Key objects need to be compared with each other for equality. If you implement IEquatable<T> you should also override the Object.Equals() method for consistency.
Override the Object.GetHashCode() method	Key objects, especially when used as keys in Hashtable and Dictionary<T Key, T Value> collections, must override the GetHashCode() method. You must also override this method if you override Object.Equals() to ensure consistent equality behavior.
Implement IComparable and IComparable<T> interfaces	If you're going to use the keys in sorted collections, the key objects must be sortable. If you don't implement these interfaces you can specify custom ordering by providing a custom comparer object.

Table 10-4: Rules For Creating Key Classes

Object Immutability

An immutable object is one whose state cannot be changed after it has been created. Strings are an example of immutable objects. One simple way to create an immutable type is to make the fields readonly and supply readonly properties. Object state values are set only through constructor methods. Care must also be taken not to return references to contained objects. Example 10.9 demonstrates this strategy.

10.9 MyImmutableType.cs

```
using System;

public class MyImmutableType {
  private readonly string _stringVal;
  private readonly int _intVal;

  public MyImmutableType(string s, int i){
    _stringVal = s;
    _intVal = i;
  }

  public string StringValue {
    get {  return string.Copy(_stringVal); }
  }

  public int IntVal {
    get {  return _intVal; }
  }

  public override string ToString(){
    return _stringVal + " " + _intVal;
  }
}
```

Referring to example 10.9 — the MyImmutableType class contains two readonly fields: one of type string named _stringVal and one of type int named _intVal. The constructor supplies the only way to set these field values. The StringValue and IntValue properties are readonly properties. (i.e., they only supply `get` operations) Note how the StringValue property returns a copy of the _stringVal field. Example 10.10 shows the MyImmutableType class in action, although there's not much going on!

10.10 MainApp.cs (Demonstrating MyImmutableType)

```
using System;

public class MainApp {
  public static void Main(){
    MyImmutableType mit = new MyImmutableType("An immutable type's state cannot be changed.", 49);
    Console.WriteLine(mit);
  }
} // end Main
```

Figure 10-6 shows the results of running this program.

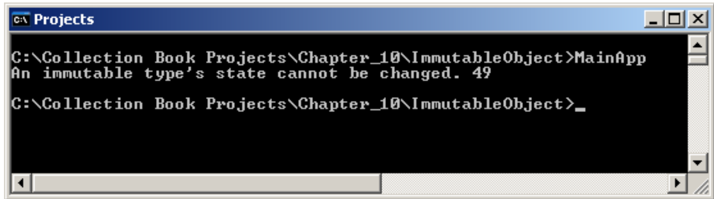


Figure 10-6: Results of Running Example 10.10

Example: PersonKey Class

Example 10.11 gives an extended key class example in the form of the PersonKey class. The PersonKey class implements most of the rules listed in table 10-4. Note that if I wanted to use this key class in sorting operations I would need to implement the IComparable and IComparable<T> interfaces.

10.11 PersonKey.cs

```
using System;

public class PersonKey : IEquatable<String> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

}
```

Referring to example 10.11 — the PersonKey class implements the IEquatable<T> interface (as IEquatable<string>). It also overrides the Object.ToString(), Object.Equals() and Object.GetHashCode() methods. It's also immutable, as the only way to set the _keyString field value is via the constructor.

Example 10.12 gives a modified version of the Person class that contains a new Key property of type PersonKey.

10.12 Person.cs (With Key Property)

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

  public Person(String firstName, String middleName, String lastName,
```



```
                    Sex gender, DateTime birthday, Guid dna){
        FirstName = firstName;
        MiddleName = middleName;
        LastName = lastName;
        Gender = gender;
        Birthday = birthday;
        DNA = dna;
     }

     public Person(String firstName, String middleName, String lastName,
                    Sex gender, DateTime birthday){
        FirstName = firstName;
        MiddleName = middleName;
        LastName = lastName;
        Gender = gender;
        Birthday = birthday;
        DNA = Guid.NewGuid();
     }

     public Person(Person p){
        FirstName = p.FirstName;
        MiddleName = p.MiddleName;
        LastName = p.LastName;
        Gender = p.Gender;
        Birthday = p.Birthday;
        DNA = p.DNA;
     }

     // public properties
     public String FirstName {
       get { return _firstName; }
       set { _firstName = value; }
     }

     public String MiddleName {
       get { return _middleName; }
       set { _middleName = value; }
     }

     public String LastName {
       get { return _lastName; }
       set { _lastName = value; }
     }

     public Sex Gender {
       get { return _gender; }
       set { _gender = value; }
     }

     public DateTime Birthday {
       get { return _birthday; }
       set { _birthday = value; }
     }

     public Guid DNA {
       get { return _dna; }
       set { _dna = value; }
     }

     public int Age {
       get {
        int years = DateTime.Now.Year - _birthday.Year;
        int adjustment = 0;
        if(DateTime.Now.Month < _birthday.Month){
            adjustment = 1;
       }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
                adjustment = 1;
       }
        return years - adjustment;
       }
     }

     public String FullName {
       get { return FirstName + " " + MiddleName + " " + LastName; }
     }

     public String FullNameAndAge {
       get { return FullName + " " + Age; }
     }

     protected String SortableName {
```



```
103         get {  return LastName + FirstName + MiddleName; }
104       }
105
106       public PersonKey Key {
107         get {  return new PersonKey(this.ToString()); }
108       }
109
110       public override String ToString(){
111         return (FullName + "  " + Gender + "  " + Age + " " + DNA);
112       }
113
114       public override bool Equals(object o){
115         if(o == null) return false;
116         if(typeof(Person) != o.GetType()) return false;
117         return this.ToString().Equals(o.ToString());
118       }
119
120       public override int GetHashCode(){
121         return this.ToString().GetHashCode();
122       }
123
124       public static bool operator ==(Person lhs, Person rhs){
125         return lhs.Equals(rhs);
126       }
127
128       public static bool operator !=(Person lhs, Person rhs){
129         return !(lhs.Equals(rhs));
130       }
131
132       public int CompareTo(object obj){
133         if((obj == null) || (typeof(Person) != obj.GetType()))  {
134           throw new ArgumentException("Object is not a Person!");
135         }
136         return this.SortableName.CompareTo(((Person)obj).SortableName);
137       }
138
139       public int CompareTo(Person p){
140         if(p == null){
141           throw new ArgumentException("Cannot compare null objects!");
142         }
143         return this.SortableName.CompareTo(p.SortableName);
144       }
145     } // end Person class
```

Referring to example 10.12 — the Key property is defined on line 106. Note that a new instance of PersonKey is returned each time the Key property is accessed. Example 10.13 demonstrates how Person objects can be inserted into a Dictionary<T Key, T Value> collection with the help of the PersonKey key class.

10.13 MainApp.cs (Demonstrating the use of Person.Key Property with a Dictionary)

```
using System;
using System.Collections.Generic;


public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
                new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
                new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
                new DateTime(1974, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
                new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
                new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
                new DateTime(1986, 10, 15), Guid.NewGuid());

    Dictionary<PersonKey, Person> directory = new Dictionary<PersonKey, Person>();
    directory.Add(p1.Key, p1);
    directory.Add(p2.Key, p2);
    directory.Add(p3.Key, p3);
    directory.Add(p4.Key, p4);
    directory.Add(p5.Key, p5);
    directory.Add(p6.Key, p6);

    foreach(KeyValuePair<PersonKey, Person> kvp in directory){
      Console.WriteLine("Key: {0}  Value: {1}", kvp.Key, kvp.Value.FullName);
    }
  }
}
```


Referring to example 10.13 — each of the six person objects created on lines 7 through 18 are inserted into the dictionary using their Key properties. The `foreach` statement on line 28 iterates over the dictionary collection and writes the value of each key and its associated value to the console.

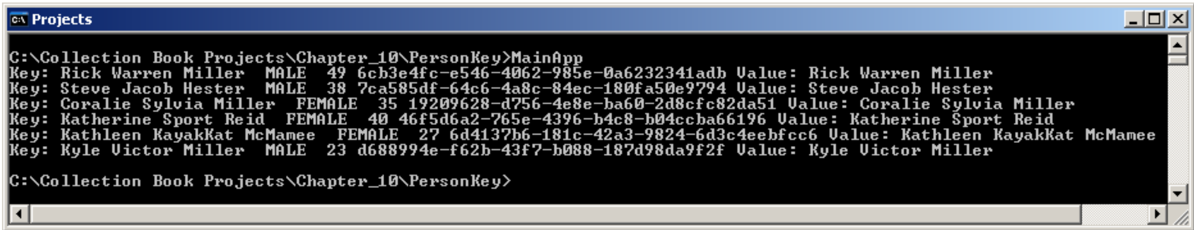


Figure 10-7: Results of Running Example 10.12

Quick Review

If an object is to be used as a key in a collection it must be immutable while it is being used as a key. Immutable object state value cannot be changed after the object is created. Key objects must also implement the IEquatable<T> interface and override the Object.Equals() and Object.GetHashCode() methods. Strings make ideal keys because they implement all the necessary interfaces and are immutable.

Summary

The first step in getting your user-defined types to behave well in collections is to override the Object.Equals() and Object.GetHashCode() methods. Make sure you adhere to the Object.Equals() method behavior rules. You can optionally overload the == and != methods as their behavior can be easily implemented in terms of the Object.Equals() method.

The overridden Object.GetHashCode() method be easily implemented by calling the GetHashCode() method on the string returned by the object's overridden ToString() method.

To specify a natural ordering for user-defined types, implement both the IComparable and IComparable<T> interfaces. To specify a custom ordering, create a custom comparer class by implementing the IComparer and IComparer<T> interfaces. It's a good idea to always implement both the generic and non-generic versions of these interfaces. Doing so ensures your user-defined types will be sortable in generic and non-generic collections.

If an object is to be used as a key in a collection it must be immutable while it is being used as a key. Immutable object state value cannot be changed after the object is created. Key objects must also implement the IEquatable<T> interface and override the Object.Equals() and Object.GetHashCode() methods. Strings make ideal keys because they implement all the necessary interfaces and are immutable.

References

Joshua Bloch. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, MA. ISBN: 0-201-31005-8.

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Derek Ashmore. *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications. DVT Press, Lombard, IL. ISBN: 0972954899*

Notes

Chapter 11


Yashica Mat 124G
Austin The Amazing Wonder Lab

Sorted Collections

Learning Objectives

- Understand how sorted collections order elements upon insertion
- Describe the operation of a red-black tree data structure
- Insert elements into a sorted collection using a custom comparer
- Use the SortedDictionary<TKey, TValue> class in a program
- Use the SortedList<TKey, TValue> class in a program
- Iterate over the elements of a sorted collection using the foreach statement
- List the required behaviors of a key class used in sorted collections

Introduction

How long does it take you to find something you're looking for? I'd say, without ever meeting you, that the answer depends on how organized you are.

Consider for a moment a messy room. It's easy to put things away! You simply walk in and throw your stuff anywhere. But finding something in the mess may take a while. You'll need to move your dirty laundry to find your car keys.

Now consider the clean, organized room. Sure, it takes a little more effort and discipline to put something in its designated spot, but you'll always know where to find it quickly when you need it.

Sorted collections are like the clean, organized room. They store their items in sorted order upon insertion. It takes a little more effort to keep the items in sorted order, but locating an item when you need it takes no time at all.

In this chapter I'll introduce you to two sorted collection types: the SortedDictionary<TKey, TValue> and the SortedList<TKey, TValue>. The SortedDictionary stores its items in a special type of binary tree called a Red-Black tree. I'll devote an entire section explaining the operation of a red-black tree and give you an extended example showing how these complex data structures work underneath the covers.

The SortedList stores its items in an array. I've covered arrays in detail in chapter 3 so I won't repeat that material here.

Upon completion of this chapter you'll have a deeper understanding of how sorted collections work and a good working example of a red-black tree, which might come in handy someday, you never know!

Red-Black Tree

A red-black tree is a special type of binary tree whose nodes contain a special attribute called Color, which can be either Red or Black. Figure 11-1 offers a representation of a red-black tree node.

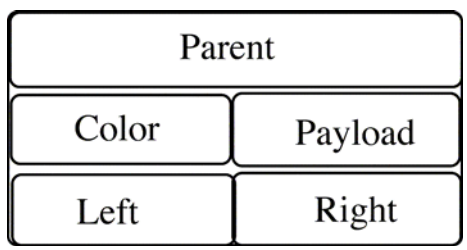


Figure 11-1: Red-Black Tree Node

Referring to figure 11-1 — the red-black tree node contains three references to other nodes: Parent, Left, and Right. The Payload reference points to the contents of the node. The Color attribute, which can be either red or black, is used to balance the tree during tree rebalancing operations. You'll see an example of this shortly.

Why Red-Black Trees?

Ordinary binary trees, that is, binary trees that do not rebalance themselves as items are inserted, can grow lopsided and degrade into a linked list if the items being inserted are already sorted. Figure 11-2 shows an example of a lopsided binary tree. Referring to figure 11-2 — in this illustration, the numbers 1 through 16 were inserted into a non-balancing binary tree in sorted, ascending order. Since the tree does not balance itself, the next inserted item is always inserted to the right of the last inserted item and the tree grows even more lopsided and degrades to the point where it's nothing more than a linked list. The problem with a linked list is that to find the last item on the list you must start at the beginning of the list, in this case the root node, and examine each item until you find the one you're looking for.

A red-black tree will examine itself after the insertion of each item and rebalance itself if necessary. Figure 11-3 shows a red-black tree after the same sequence of sixteen numbers have been inserted into the tree.

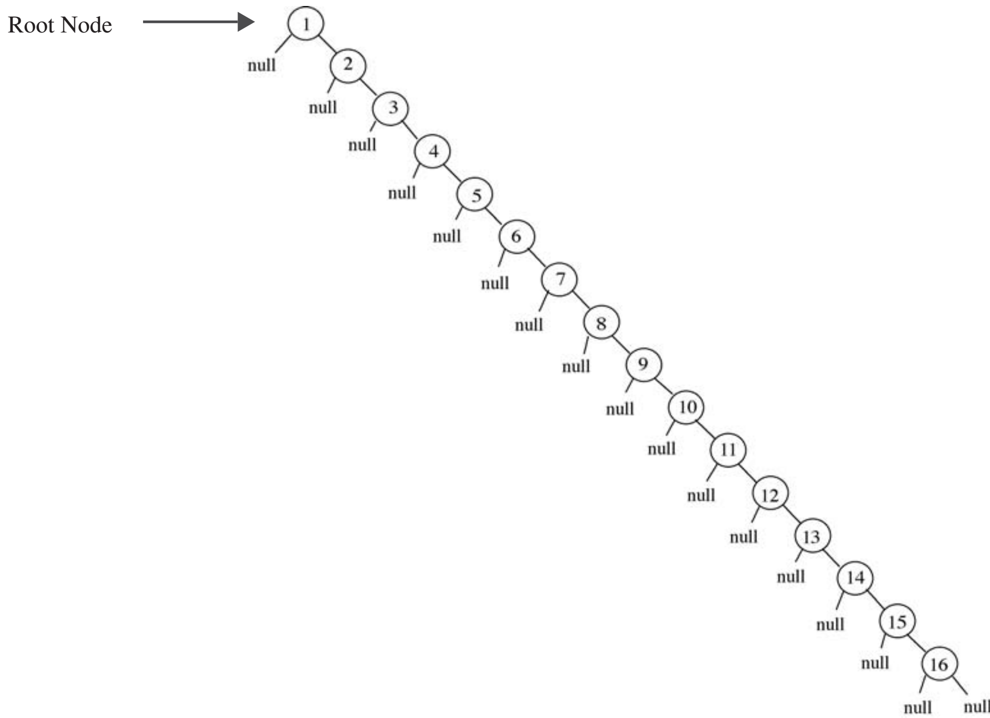


Figure 11-2: Extremely Unbalanced Binary Tree After Inserting Sixteen Sorted Numbers

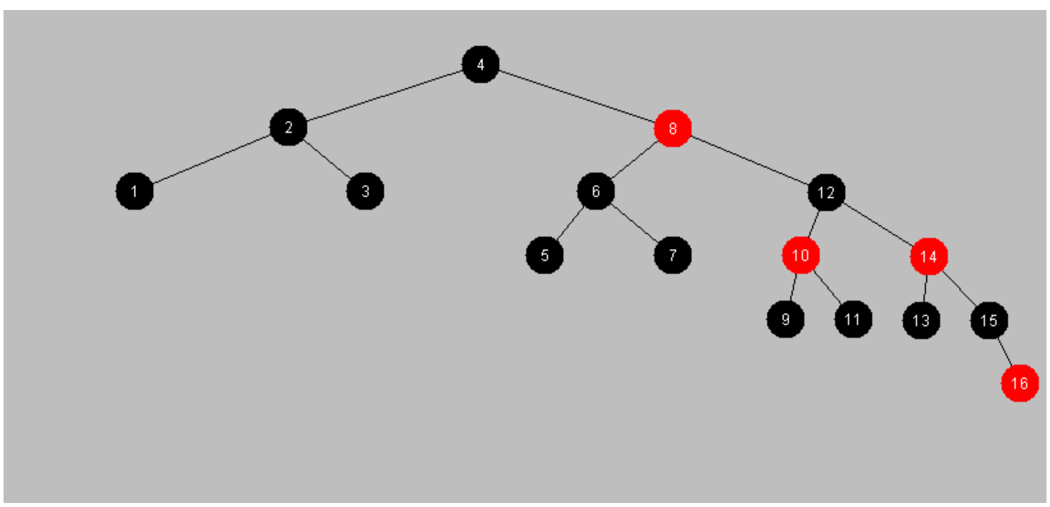


Figure 11-3: Red-Black Tree After Inserting Sixteen Sorted Numbers

As figure 11-3 illustrates, the red-black tree is in a more balanced state (although not perfectly balanced) after ingesting sixteen sorted numbers. Still, given a large set of numbers, an imperfectly balanced tree will offer better insertion and retrieval performance than a non-balanced tree. Let's take a look now at how the red-black tree operates.

Red-Black Tree Operation

In this section I'll step through the insertion of the following sequence of numbers into a red-black tree: 193, 170, 184, 35, 154, 134, 47, 119, 171, 104, 161, 104, 77, 5, 57, 168, 31, 82, 75, 62. This sequence of numbers was generated

with a random number generator — the Random class in the .NET framework. You may have noticed the number 104 occurred twice. That's not a mistake, that's just the sequence as generated for this particular example.

The Rules of the Game

A red-black tree will maintain balance by enforcing the following properties or constraints:

1: Each node in the tree is either red or black
2: All new nodes are inserted with the color red
3: The root node is always black
4: Every null leaf node is considered black
5: If a node is red then both its children nodes are black
6: All paths from a particular node to the root node contain the same number of black nodes

To maintain the above listed constraints, a node, upon insertion, may have to be moved into a new position via an operation referred to as a *rotate* of which there are two types: *RotateRight* and *RotateLeft*. These rotate operations are performed in conjunction with the operation that enforces the above listed constraints.

Now, if you're like me, and this explanation of red-black tree constraints has left you scratching your head in confusion, then a few pictures are in order. Actually, even better, open your web browser and go to the following site: [http://gauss.ececs.uc.edu/RedBlack/redblack.html] and insert the numbers in the sequence listed at the beginning of this section and watch the animation to see how the tree rebalances itself.

Figure 11-4 shows the state of a red-black tree after the insertion of the first three numbers 193, 170, and 184. The first number, 193, is inserted as a red node initially, but it immediately becomes the root node and is colored black. The second number, 170 is less than 193 and is inserted to the left of the root node. It is inserted red and stays red.

The third number, 184, is less than the root node but greater than 170. It is inserted to the right of 170 and is red upon insertion. The insertion operation then inspects its parent node and finding it is red must fix the tree according to the following cases:

Case 1: The new node's uncle node is red.
Case 2: The new node's uncle node is black and the new node is a right child
Case 3: The new node's uncle node is black and the new node is a left child

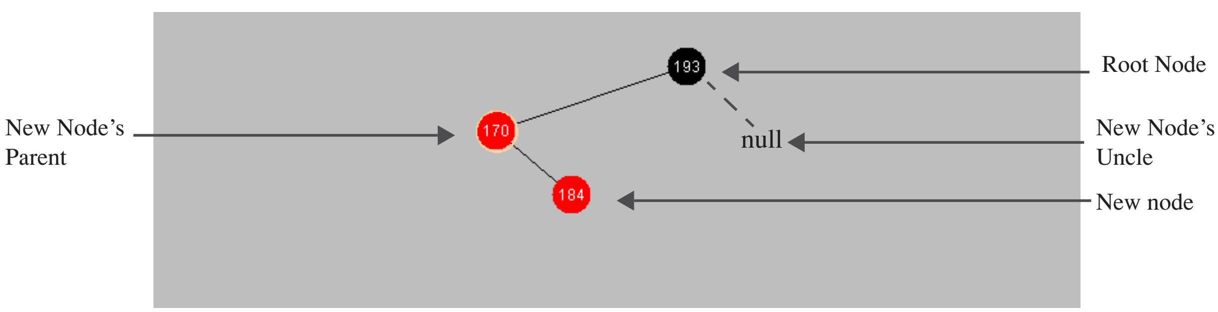


Figure 11-4: State of Red-Black Tree after Inserting numbers 193, 170, & 184

Referring to figure 11-4 — with the exception of the root node, all newly inserted nodes have a parent node. In this case the node numbered 170 is the parent of node 184. The uncle of node 184, in this situation, is the right child of the root node. However, since there is nothing yet inserted to the right of the root node, the root node's Right reference points to null which is considered to be colored black. And since node 184 is a right child node, case 2 applies which will trigger the following sequence of events:

1: A RotateLeft operation will be performed around node 170. This will move node 184 into node 170's position, making it the left child of node 193, and make node 170 the left child of node 184.

2: A RotateRight will be performed around node 193 which will make node 184 the new root node. Node 193 will be set to Red.

3: Since node 184 is now the root node, it will be set to Black.

Figures 11-5 and 11-6 show the results of performing RotateLeft and RotateRight on nodes in general. Figures 11-7 through 11-9 show how the tree in question will look when the rotate and recoloring operations are applied to rebalance the tree.

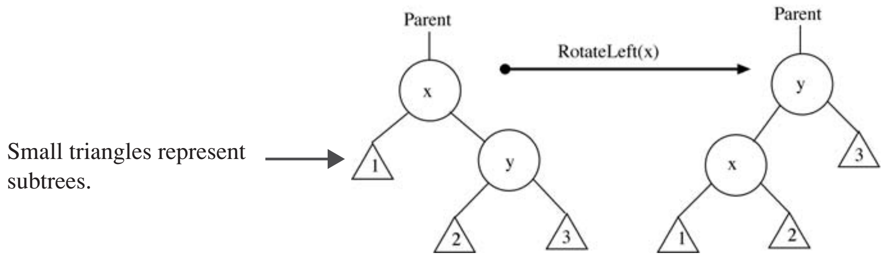


Figure 11-5: RotateLeft Operation

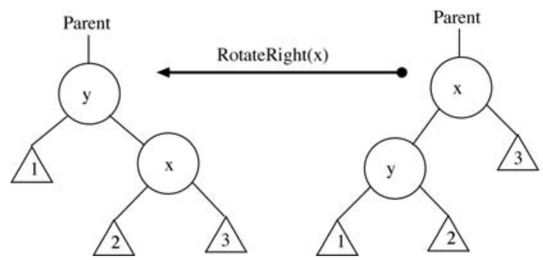


Figure 11-6: RotateRight Operation

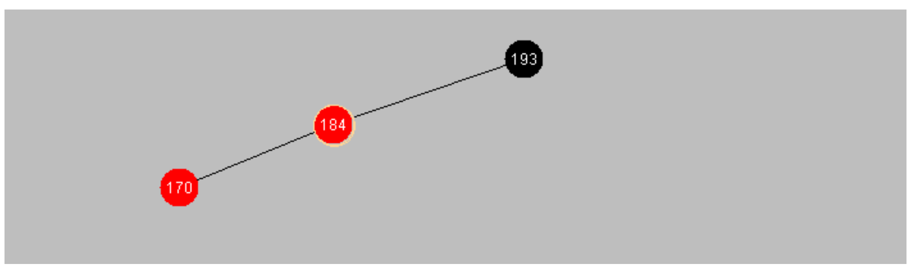


Figure 11-7: After RotateLeft — Node 184 is now the Parent of Node 170 and the Left Child of Node 193

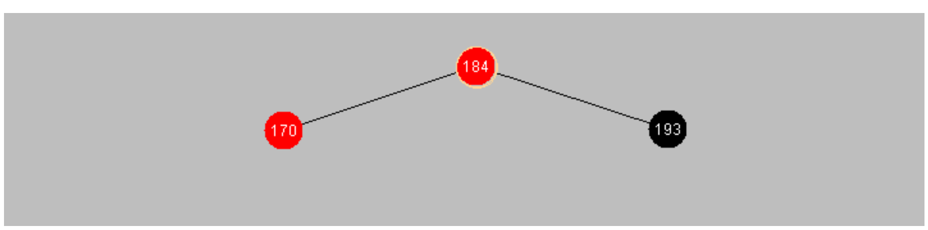


Figure 11-8: After RotateRight — Node 184 is the New Root Node

Upon insertion of all the numbers in the sequence the tree will look like figure 11-10.

Referring to figure 11-10 — during the course of inserting the full sequence of numbers, a series of rebalancing and recoloring operations are performed to bring the red-black tree into compliance with the constraints listed earlier in this section.

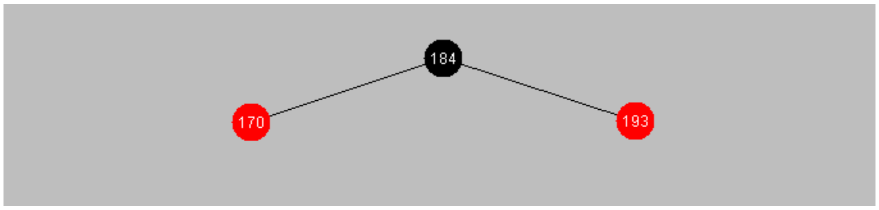


Figure 11-9: Node 184 is Set to Black. Node 193 is Set to Red. The Tree is now Balanced.

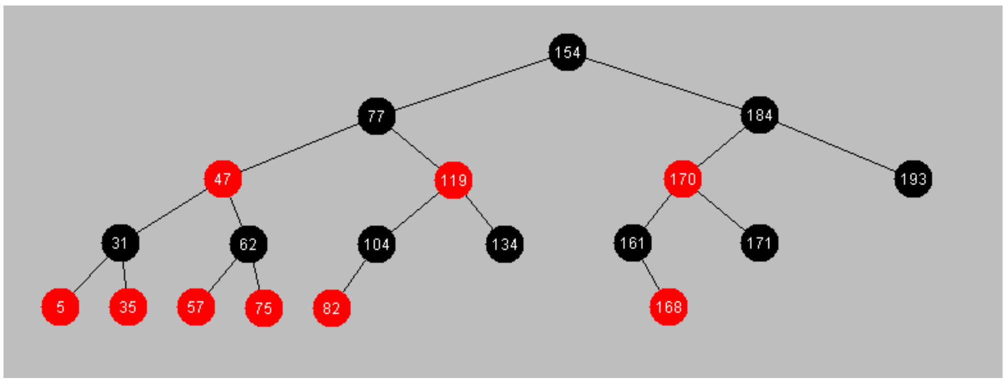


Figure 11-10: Final State of the Red-Black Tree after Inserting all Numbers in the Sequence

Red-Black Tree Code

This section gives a complete example of a red-black tree. I'll start with the KeyValuePair<TKey, TValue> class.

11.1 KeyValuePair.cs

```
using System;

public class KeyValuePair<TKey, TValue> : IComparable<KeyValuePair<TKey, TValue>> where TKey :
                                                                                 IComparable<TKey> {

  private TKey _key;
  private TValue _value;

  public KeyValuePair(TKey key, TValue value) {
    _key = key;
    _value = value;
  }

  public KeyValuePair() { }

  public TKey Key {
    get { return _key; }
    set { _key = value; }
  }

  public TValue Value {
    get { return _value; }
    set { _value = value; }
  }

  public int CompareTo(KeyValuePair<TKey, TValue> other) {
    return this._key.CompareTo(other.Key);
  }

  public override string ToString() {
    return _key.ToString() + " " + _value.ToString();
  }
} // End KeyValuePair class definition
```


Referring to example 11.1 — the KeyValuePair<TKey, TValue> class places a constraint on the TKey generic parameter specifying that whatever type of key is used it must implement the IComparable<TKey> interface. (**Note:** *All the built-in C#.NET types already implement both the IComparable and IComparable<T> interfaces*.)

Example 11.2 gives the code for the Node<TKey, TValue> class.

11.2 Node.cs

```
using System;

public class Node<TKey, TValue> where TKey : IComparable<TKey> {

  public KeyValuePair<TKey, TValue> Payload;
  public Node<TKey, TValue> Parent;
  public Node<TKey, TValue> Left;
  public Node<TKey, TValue> Right;

  private bool _color;
  private const bool RED = true;
  private const bool BLACK = false;


  public Node(KeyValuePair<TKey, TValue> payload) {
    Payload = payload;
    _color = RED;
  }

  public bool IsRed {
    get { return _color; }
  }

  public bool IsBlack {
    get { return !IsRed; }
  }

  public void MakeRed() {
    _color = RED;

  }

  public void MakeBlack() {
    _color = BLACK;

  }

  public string Color {
    get { return (_color == RED) ? "RED" : "BLACK"; }
    set {
      switch (value) {
        case "RED": _color = true;
          break;
        case "BLACK": _color = false;
          break;
      }
    }
  }
}
```

Referring to example 11.2 — the Node<TKey, TValue> class places a constraint on the TKey generic parameter specifying that whatever type is supplied must implement the IComparable<TKey> interface. It contains four public fields: Payload, which is a reference to the value supplied to the node which is of type KeyValuePair<TKey, TValue>, and Parent, Left, and Right, which are references to the parent, left, and right nodes respectively. The class contains a private field named _color which is of type bool. It declares two bool constants: RED and BLACK which are true and false respectively. The Node class also defines three properties: IsRed, IsBlack, and Color, which provides a string representation of a node's color attribute. The MakeRed() and MakeBlack() methods are self explanatory.

Example 11.3 lists the code for the RedBlackTree class. The code you see here is a straightforward implementation of the red-black tree algorithms found in *Introduction to Algorithms, Second Edition*, by Thomas H. Cormen, et. al., MIT Press. (*See References section for complete citation*.) My implementation differs from their algorithms in that I used null references to indicate empty leaf nodes vs. the single nil field. This increased the complexity of my code somewhat but only in that before proceeding with an operation the code must ensure a reference is not null.

11.3 RedBlackTree.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public class RedBlackTree<TKey, TValue> : IEnumerable where TKey : IComparable<TKey> {


  #region Constants
  private const int EQUALS = 0;
  private const int LESSTHAN = -1;
  private const int GREATERTHAN = 1;
  #endregion

  #region Fields
  private Node<TKey, TValue> _root;
  private int _count = 0;
  private int _left_rotates = 0;
  private int _right_rotates = 0;
  private TKey _first_inserted_key;
  private bool _debug = true;
  #endregion


  #region Constructors
  public RedBlackTree() : this(true) { }

  public RedBlackTree(bool debug) {
    _debug = debug;
  }
  #endregion


  #region Properties
  public KeyValuePair<TKey, TValue> Root {
    get { return _root.Payload; }
  }

  public int Count {
    get { return _count; }
  }
  #endregion


  #region Methods


  /****************************************************************
   *   Insert Method
   * ************************************************************/
  public void Insert(TKey key, TValue value) {
    if ((key == null) || (value == null)) {
      throw new ArgumentException("Invalid Key and/or Value arguments!");
    }
    if (_root == null) {
      _root = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

      _count++;
      if (_debug) {
        Console.WriteLine("Inserted root node with values:" + _root.Payload.ToString());
      }
      _root.MakeBlack();
      _first_inserted_key = _root.Payload.Key;
      return;
    } else {
      Node<TKey, TValue> new_node = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

      bool inserted = false;
      int comparison_result = 0;
      Node<TKey, TValue> node = _root;
      while (!inserted) {
        comparison_result = new_node.Payload.Key.CompareTo(node.Payload.Key);
        switch (comparison_result) {
          case EQUALS: inserted = true; // ignore duplicate key values
            break;
          case LESSTHAN: if (node.Left == null) {
              node.Left = new_node;
              new_node.Parent = node;
              inserted = true;
```



```
                _count++;
                if (_debug) {
                  Console.WriteLine("Inserted left: {0}", new_node.Payload.Key);
                }
                RBInsertFixUp(new_node);

              } else {
                node = node.Left;
              }
              break;
            case GREATERTHAN: if (node.Right == null) {
                node.Right = new_node;
                new_node.Parent = node;
                inserted = true;
                _count++;
                if (_debug) {
                  Console.WriteLine("Inserted right: {0}", new_node.Payload.Key);
                }
                RBInsertFixUp(new_node);
              } else {
                node = node.Right;
              }
              break;
          }
        }
      }
    } // end Insert() method




    /****************************************************************
     *  RBInsertFixUp Method
     * ************************************************************/
    private void RBInsertFixUp(Node<TKey, TValue> node) {
      while ((node.Parent != null) && (node.Parent.IsRed)) {
        Node<TKey, TValue> y = null;
        if ((node.Parent.Parent != null) && (node.Parent == node.Parent.Parent.Left)) {
                                                        // Parent is a left child
          y = node.Parent.Parent.Right;

          if ((y != null) && (y.IsRed)) {   //case 1
            node.Parent.MakeBlack();        //case 1
            y.MakeBlack();                  //case 1
            node.Parent.Parent.MakeRed();   //case 1
            node = node.Parent.Parent;

            if (node.IsRed) {
              continue;
            }

          } else if (node == node.Parent.Right) {  //case 2
            node = node.Parent;                    //case 2
            RotateLeft(node);                      //case 2
          }

          /**************/
          if ((node.Parent != null)) {           //case 3
            node.Parent.MakeBlack();             //case 3
            if (node.Parent.Parent != null) {    //case 3
              node.Parent.Parent.MakeRed();      //case 3
              RotateRight(node.Parent.Parent);   //case 3
            }
          }
          /*******************/

        } else {                                                //Parent is a right child

          if (node.Parent.Parent != null) {
            y = node.Parent.Parent.Left;
          }

          if ((y != null) && (y.IsRed)) {       //case 1
            node.Parent.MakeBlack();            //case 1
            y.MakeBlack();                      //case 1
            node.Parent.Parent.MakeRed();       //case 1
            node = node.Parent.Parent;

            if (node.IsRed) {
              continue;
            }
```



```
          } else if (node == node.Parent.Left) {   //case 2
            node = node.Parent;                    //case 2
            RotateRight(node);                     //case 2
          }

          /*******************/
          if ((node.Parent != null)) {             //case 3
            node.Parent.MakeBlack();               //case 3
            if (node.Parent.Parent != null) {      //case 3
              node.Parent.Parent.MakeRed();        //case 3
              RotateLeft(node.Parent.Parent);      //case 3
            }
          }
          /********************/

        } // end if

        _root.MakeBlack();

      } // end while
    } // end RBInsertFixUp() method




    /**************************************************************
     * RotateLeft Method
     * ***********************************************************/
    private void RotateLeft(Node<TKey, TValue> x) {
      if (x.Right != null) {
        if (_debug) {
          Console.WriteLine("*******************************************************");
          Console.WriteLine("Left Rotate tree  with node x = {0}", x.Payload.Key);
          Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                           (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
          Console.WriteLine("*******************************************************");
        }

        Node<TKey, TValue> y = x.Right;
        if (y != null) {
          x.Right = y.Left;
          if (y.Left != null) {
            y.Left.Parent = x;
          }
          y.Parent = x.Parent;
          if (x.Parent == null) {
            _root = y;
          } else if (x == x.Parent.Left) {
            x.Parent.Left = y;
          } else {
            x.Parent.Right = y;
          }

          y.Left = x;
          x.Parent = y;
        }

        _left_rotates++;
        if (_debug) {
          Console.WriteLine("*******************************************************");
          Console.WriteLine("After left rotate node x = {0}", x.Payload.Key);
          Console.WriteLine("Node x parent = {0}",
                           (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
          Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                           (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
          Console.WriteLine("Left Rotates: {0}", _left_rotates);
          Console.WriteLine("*******************************************************");
        }
      }
    } // end RotateLeft() method



    /**************************************************************
     *  RotateRight Method
     * ***********************************************************/
    private void RotateRight(Node<TKey, TValue> x) {
      if (x.Left != null) {
        if (_debug) {
          Console.WriteLine("*******************************************************");
```

```
          Console.WriteLine("Right Rotate tree  with node x = {0}", x.Payload.Key);
          Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                            (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
          Console.WriteLine("******************************************************");
        }

        Node<TKey, TValue> y = x.Left;
        if (y == null) {
          x.Left = null;
        } else {
          x.Left = y.Right;
          if (y.Right != null) {
            y.Right.Parent = x;
          }
          y.Parent = x.Parent;
          if (y.Parent == null) {
            _root = y;
          } else if (x == y.Parent.Left) {
            y.Parent.Left = y;
          } else {
            y.Parent.Right = y;
          }
          y.Right = x;
          x.Parent = y;
        }

        _right_rotates++;
        if (_debug) {
          Console.WriteLine("******************************************************");
          Console.WriteLine("After right rotate node x = {0}", x.Payload.Key);
          Console.WriteLine("Node x parent = {0}",
                            (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
          Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                            (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
          Console.WriteLine("Root = {0}", _root.Payload.Key);
          Console.WriteLine("Right Rotates: {0}", _right_rotates);
          Console.WriteLine("******************************************************");
        }

      }
    } // end RotateRight() method


    /***************************************************************
     *  Search Method
     * ************************************************************/
    public Node<TKey, TValue> Search(TKey key) {
      int compare_result = 0;
      bool key_found = false;
      Node<TKey, TValue> node = _root;

      while (!key_found) {
        compare_result = key.CompareTo(node.Payload.Key);
        switch (compare_result) {
          case EQUALS: key_found = true;
            break;
          case LESSTHAN: if (node.Left == null) {
              return null;
            }
            node = node.Left;
            break;
          case GREATERTHAN: if (node.Right == null) {
              return null;
            }
            node = node.Right;
            break;

        }
      }

      return node;
    }


    /***************************************************************
     *  Delete Method
     * ************************************************************/
    public void Delete(Node<TKey, TValue> z) {
      if (z == null) return;
      Node<TKey, TValue> y = null;
      if ((z.Left == null) || (z.Right == null)) {
```

```
          y = z;
        } else {
          y = TreeSuccessor(z);
        }

        Node<TKey, TValue> x = null;

        if (y.Left != null) {
          x = y.Left;
        } else {
          x = y.Right;
        }
        if (x != null) {
          x.Parent = y.Parent;
        }

        if (y.Parent == null) {
          _root = x;
        } else if (y == y.Parent.Left) {
          y.Parent.Left = x;
        } else {
          y.Parent.Right = x;
        }

        if (y != z) {
          z.Payload = y.Payload;
        }

        if (y.IsBlack) {
          RBDeleteFixUp(x);
        }
        _count--;
      }



      /****************************************************************
       *  RBDeleteFixup
       * **************************************************************/
      private void RBDeleteFixUp(Node<TKey, TValue> x) {
        while ((x != null) && (x != _root) && (x.IsBlack)) {
          if (x == x.Parent.Left) {
            Node<TKey, TValue> w = x.Parent.Right;
            if ((w != null) && w.IsRed) {
              w.MakeBlack();
              x.Parent.MakeRed();
              RotateLeft(x.Parent);
              w = x.Parent.Right;
            }

            if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                            && ((w.Right == null) || w.Right.IsBlack)) {
              w.MakeRed();
              x = x.Parent;
              continue;

            } else if ((w != null) && w.Right.IsBlack) {
              w.Left.MakeBlack();
              w.MakeRed();
              RotateRight(w);
              w = x.Parent.Right;
            }

            /************************/
            if (w != null) {
              w.Color = x.Parent.Color;
              x.Parent.MakeBlack();
              w.Right.MakeBlack();

            }
            RotateLeft(x.Parent);
            x = _root;
            /***********************/

          } else {

            Node<TKey, TValue> w = x.Parent.Left;
            if ((w != null) && w.IsRed) {
              w.MakeBlack();
              x.Parent.MakeRed();
              RotateRight(x.Parent);
```



```
            w = x.Parent.Left;
          }

          if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                          && ((w.Right == null) || w.Right.IsBlack)) {
            w.MakeRed();
            x = x.Parent;
            continue;

          } else if ((w != null) && w.Left.IsBlack) {
            w.Right.MakeBlack();
            w.MakeRed();
            RotateLeft(w);
            w = x.Parent.Left;
          }

          /***********************/
          if (w != null) {
            w.Color = x.Parent.Color;
            x.Parent.MakeBlack();
            w.Right.MakeBlack();

          }
          RotateRight(x.Parent);
          x = _root;
          /**********************/
        }

        x.MakeBlack();
        _root.MakeBlack();
      } // end while

    } // end RBDeleteFixUp



    /***************************************************************
     * TreeSuccessor Method
     * ***********************************************************/
    private Node<TKey, TValue> TreeSuccessor(Node<TKey, TValue> node) {
      if (node.Right != null) {
        return TreeMinimum(node.Right);
      }
      Node<TKey, TValue> y = node.Parent;
      while ((y != null) && (node == y.Right)) {
        node = y;
        y = y.Parent;
      }
      return y;
    }


    /***************************************************************
     * TreeMinimum Method
     * ***********************************************************/
    private Node<TKey, TValue> TreeMinimum(Node<TKey, TValue> node) {
      while (node.Left != null) {
        node = node.Left;
      }
      return node;
    }


    /***************************************************************
     *  TreeMaximum Method -- Not used in this program
     * ***********************************************************/
    private Node<TKey, TValue> TreeMaximum(Node<TKey, TValue> node) {
      while (node.Right != null) {
        node = node.Right;
      }
      return node;
    }


    /***************************************************************
     *  GetEnumerator Method
     * ***********************************************************/
    public IEnumerator GetEnumerator() {
      return this.ToArray().GetEnumerator();
    }

```



```
      /************************************************************
       *  ToArray Method
       * **********************************************************/
      public KeyValuePair<TKey, TValue>[] ToArray() {
        KeyValuePair<TKey, TValue>[] _items = new KeyValuePair<TKey, TValue>[_count];
        int index = 0;
        this.WalkTree(_root, _items, ref index);
        return _items;

      }


      /************************************************************
       *  WalkTree Method
       * **********************************************************/
      private void WalkTree(Node<TKey, TValue> node, KeyValuePair<TKey, TValue>[] items, ref int index) {
        if (node != null) {
          WalkTree(node.Left, items, ref index);
          items[index++] = node.Payload;
          if (_debug) {
            if (node == _root) {
              Console.WriteLine("**********ROOT NODE: {0}:{1}**********",
                                node.Payload.Value, node.Color);
            } else {
              Console.WriteLine("Walking Tree - Node visited: {0} Color: {1}",
                                node.Payload.Value, node.Color);
            }
          }
          WalkTree(node.Right, items, ref index);
        }
      }


      /************************************************************
       * PrintTreeToConsole Method
       * **********************************************************/
      public void PrintTreeToConsole() {
        foreach (KeyValuePair<TKey, TValue> item in this) {
          if (item.Key.CompareTo(_root.Payload.Key) == 0) {
            Console.ForegroundColor = ConsoleColor.Yellow;
            Console.Write(item.Key + " ");
            Console.ForegroundColor = ConsoleColor.White;
          } else {
            Console.Write(item.Key + " ");
          }
        }
        Console.WriteLine();
      }


      /************************************************************
       *  PrintTreeStats Method
       * **********************************************************/
      public void PrintTreeStats() {
        Console.WriteLine("------------ Tree Stats --------------------");
        Console.WriteLine("First inserted key: {0}", _first_inserted_key);
        Console.WriteLine("Count: {0}", _count);
        Console.WriteLine("Left Rotates: {0}", _left_rotates);
        Console.WriteLine("Right Rotates: {0}", _right_rotates);
        Console.WriteLine("--------------------------------------------");
      }

      #endregion

    } // end RedBlackTree class
```

Referring to example 11.3 — let's survey the class as a whole before diving into the details, so your brain doesn't explode. The RedBlackTree<TKey, TValue> class implements IEnumerable and places a constraint on the TKey type specifying that it must implement the IComparable<TKey> interface. Then there appear some constants and some fields. The constants are self explanatory. They're used in the body of the Insert() method. One of the fields is named _debug and is used to switch on and off some debugging code I used to help me write this code. I've left the debugging code in so you can run the code with or without debugging. You can set the debugging mode via a constructor parameter when you create an instance of RedBlackTree. The only fields that matter are _root and _count. The rest of the fields I use to keep track of various tree statistics, again for debugging purposes during development. The two properties Root and Count are self-explanatory.

Now, onto the methods. First let me say that the bulk of functionality of the RedBlackTree class is contained in the following methods: Insert(), RBInsertFixUp(), RotateLeft(), RotateRight(), RBDeleteFixUp(), TreeSuccessor() and TreeMinimum().

Another recommendation before getting started: If you want to see this code in action and trace its execution, load this code into Visual Studio, set some breakpoints, and run it in debug mode.

The Insert() method on line 51 takes a key and value and does a non-recursive tree insertion. If the _root field is null it will create a new node and make it the root. If the root is occupied, the method will compare the keys and based on the result of the comparison walk the tree to the left or right until it finds a null left or right child reference, at which point the insertion takes place.

The RBInsertFixUp() method on line 114 gets called after each non-root insertion. The purpose of the RBInsert-FixUp() method is to set the tree in order by recoloring and rotating nodes as required to maintain balance. The RBIn-sertFixUP() method does most of its work in the body of the `if/else` statement that begins on line 117. The upper part of the `if` statement applies if the inserted node's parent is a left child while the `else` part applies if the node's parent is a right child.

During the course of execution, the RBInsertFixUp() method will call either the RotateLeft() or RotateRight() methods. These methods perform the indicated rotations as illustrated in figures 11-5 and 11-6 respectively. Again, a lot of what the code in these methods does it check to ensure that a reference is not null before attempting to perform an operation on it.

The GetEnumerator() method is required as part of the IEnumerable interface. I cheated here and implemented it terms of the ToArray() method. (*Nothing wrong with cheating a little is there?*) Actually, this worked better than I expected and was straightforward to code. The ToArray() method relies on the WalkTree() method, which performs a recursive inorder walk of the tree. An inorder tree walk returns items in ascending order beginning with the minimum valued item or the left-most lowest leaf node first.

The Search() method is used to look for items in the tree. It performs a non-recursive tree walk comparing keys along the way. If it finds what it's looking for it returns the whole node, otherwise it returns null.

The Delete() method depends on the TreeSuccessor() method which depends on the TreeMinimum() method. When a node is deleted, the RBDeleteFixUp() method must be called to put the tree in sorts upon removal of the indicated node.

The MainApp class given in example 11.4 creates an instance of a RedBlackTree and puts it through its paces using the sequence of numbers listed at the beginning of this section.

11.4 MainApp.cs

```
using System;

public class MainApp {
  public static void Main(string[] args) {
    bool debugOn = false;
    if (args.Length > 0) {
      try {
        debugOn = Convert.ToBoolean(args[0]);
      } catch (Exception) {
        debugOn = false;
      }
    }

    RedBlackTree<int, int> tree = new RedBlackTree<int, int>(debugOn);
    Random random = new Random();
    int[] vals = new int[20];

    for (int i = 0; i < 20; i++) {
      vals[i] = random.Next(200);
      tree.Insert(vals[i], vals[i]);
    }

    tree.PrintTreeStats();
    Console.WriteLine("Original insertion order:");
    foreach(int i in vals){
      Console.Write(i + " ");
    }
    Console.WriteLine();
    Console.WriteLine("Sorted Order:");
    tree.PrintTreeToConsole();
  }
}
```

Referring to example 11.4 — the MainApp class can be run with an optional debug command-line argument of either "true" or "false". If the args array contains an argument, it attempts to convert the string into a boolean value,

otherwise, it sets the debug variable to false. An instance of RedBlackTree is created on line 14 followed by the creation of an instance of the Random class and an array to hold 20 integer values which are created in the `for` loop on line 18 with the help of the random.Next() method. As each random number is generated it's inserted into the array and also into the red-black tree. On line 23 a call to PrintTreeStats() prints the tree statistics to the console. The original insertion order of the randomly generated values is then printed to the console, followed by the sorted order, which is generated by a call to PrintTreeToConsole(). Figure 11-11 shows the results of running this program.

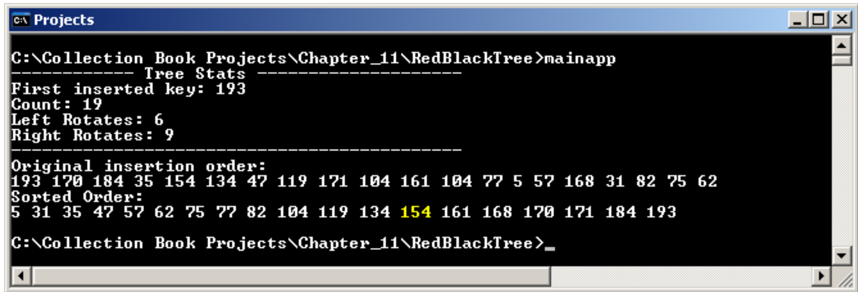


Figure 11-11: Results of Running Example 11.4

Referring to figure 11-11 — note that each time you run the application as is, you'll get different results than what are shown in this figure. That's because of the different random values being generated each time. I recommend you modify the MainApp.cs file and insert different sets of values. Delete items from the tree and note the effects insertions and deletions have on the tree. Figure 11-12 shows the application being run with the debug value "true" at the command line.

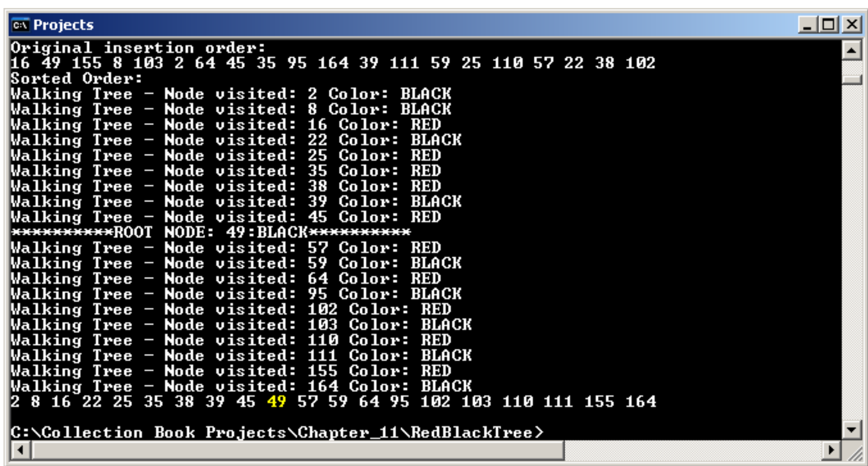


Figure 11-12: Partial Listing of Running Example 11.4 with Debug set to True at the Command Line

Quick Review

A red-black tree is a special kind of binary tree whose nodes contain an extra piece of information indicating its color which can be either red or black. The red-black tree rebalances itself when necessary after each item insertion by recoloring and rotating nodes based on a set of constraints and cases. The benefit to using a red-black tree is that it won't degrade into a linked-list when fed a list of already-sorted items.

SortedDictionary<TKey, TValue>

The SortedDictionary<TKey, TValue> class, found in the System.Collections.Generic namespace, has as its foundational data structure a red-black binary tree. Of course, since it's a collection class, it has a lot more functionality than my red-black tree data structure given in the previous section. Figure 11-13 gives a UML class diagram showing the inheritance hierarchy of the SortedDictionary<TKey, TValue> class.

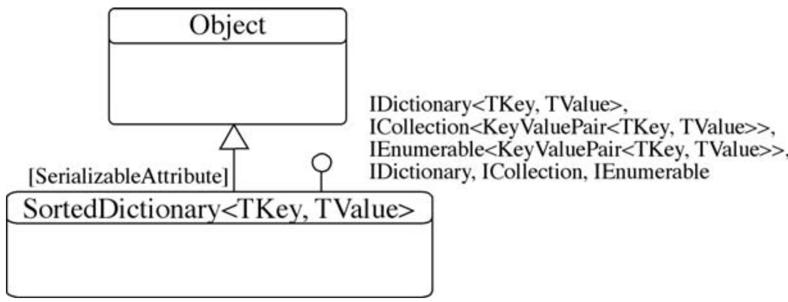


Figure 11-13: SortedDictionary<TKey, TValue> UML Class Diagram

Referring to figure 11-13 — the SortedDictionary<TKey, TValue> class implements the IDictionary<TKey, TValue>, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IDictionary, ICollection, and IEnumerable interfaces. The following sections explain the purpose of these interfaces in greater detail.

IEnumerable<KeyValuePair<TKey, TValue>> and IEnumerable Interfaces

These interfaces expose an enumerator that allows the collection to be iterated over using a `foreach` statement. You could manipulate the enumerator directly but that's not the way things are usually done. Note that you cannot alter the value of an item extracted from a collection via an enumerator. Doing so will invalidate the collection and throw an exception.

Note that the items within a SortedDictionary<TKey, TValue> collection are KeyValuePair<TKey, TValue> objects. When using the `foreach` statement to iterate over the SortedDictionary's items, you'll have to keep this in mind. Alternatively, you can extract just the SortedDictionary's keys or values via its Keys or Values properties.

ICollection<KeyValuePair<TKey, TValue>> and ICollection Interfaces

The ICollection and ICollection<KeyValuePair<TKey, TValue>> interfaces tag the SortedDictionary<TKey, TValue> class as a collection type. The ICollection interface declares object synchronization properties IsSynchronized and SyncRoot, while the ICollection<KeyValuePair<TKey, TValue>> interface declares the Add(), Remove(), and Contains() methods and the Count property. These interfaces also declare the GetEnumerator() methods required to iterate over the collection with a `foreach` statement.

IDictionary<TKey, TValue> and IDictionary Interfaces

The IDictionary and IDictionary<KeyValuePair<TKey, TValue>> interfaces provide the non-generic and generic versions of Keys and Values properties, the indexer, and the ContainsKey() and the TryGetValue() methods.

SortedDictionary<TKey, TValue> Example Program

This section presents a short example program that demonstrates the use of the SortedDictionary<TKey, TValue> collection. It uses the Person class as defined in example 11.5, the PersonKey class shown in example 11.6, and the PersonAgeComparer class shown in example 11.7. First, the Person class.

11.5 Person.cs

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex { MALE, FEMALE} ;


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;


  public Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
    get {  return _firstName; }
    set {  _firstName = value; }
  }

  public String MiddleName {
    get {  return _middleName; }
    set {  _middleName = value; }
  }

  public String LastName {
    get {  return _lastName; }
    set {  _lastName = value; }
  }

  public Sex Gender {
    get {  return _gender; }
    set {  _gender = value; }
  }

  public DateTime Birthday {
    get {  return _birthday; }
    set {  _birthday = value; }
  }

  public Guid DNA {
    get {  return _dna; }
    set {  _dna = value; }
  }

```

```
    public int Age {
      get {
        int years = DateTime.Now.Year - _birthday.Year;
        int adjustment = 0;
        if(DateTime.Now.Month < _birthday.Month){
           adjustment = 1;
        }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
                 adjustment = 1;
         }
        return years - adjustment;
      }
    }

    public String FullName {
      get { return FirstName + " " + MiddleName + " " + LastName; }
    }

    public String FullNameAndAge {
      get { return FullName + " " + Age; }
    }

    protected String SortableName {
      get { return LastName + FirstName + MiddleName; }
    }

    public PersonKey Key {
      get { return new PersonKey(this.ToString()); }
    }

    public override String ToString(){
      return (FullName + "  " + Gender + "  " + Age + " " + DNA);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(Person) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public static bool operator ==(Person lhs, Person rhs){
      return lhs.Equals(rhs);
    }

    public static bool operator !=(Person lhs, Person rhs){
      return !(lhs.Equals(rhs));
    }

    public int CompareTo(object obj){
      if((obj == null) || (typeof(Person) != obj.GetType()))  {
        throw new ArgumentException("Object is not a Person!");
      }
      return this.SortableName.CompareTo(((Person)obj).SortableName);
    }

    public int CompareTo(Person p){
      if(p == null){
        throw new ArgumentException("Cannot compare null objects!");
      }
      return this.SortableName.CompareTo(p.SortableName);
    }
  } // end Person class
```

Referring to example 11.5 — the Person class is fairly self-explanatory. The only property of note is the Key property, which returns a PersonKey object. The PersonKey class is given in example 11.6.

11.6 PersonKey.cs

```
using System;

public class PersonKey : IEquatable<String>, IComparable, IComparable<PersonKey> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public bool Equals(string other){
```

```
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public int CompareTo(object obj){
     return _keyString.CompareTo(obj);
    }


    public int CompareTo(PersonKey pk){
      return _keyString.CompareTo(pk._keyString);
    }
}
```

Referring to example 11.6 — the PersonKey class implements the IEquatable, IComparable, and IComparable<PersonKey> interfaces. The IComparable and IComparable<PersonKey> interfaces allow objects of type PersonKey to be used in sorted collections where keys are compared with each other to determine sorted order.

11.7 PersonAgeComparer.cs

```
using System;
using System.Collections;
using System.Collections.Generic;

public class PersonAgeComparer : IComparer, IComparer<Person> {

  public int Compare(object x, object y){
    if((x == null) || (y == null) || (typeof(Person) != x.GetType())
                  || (typeof(Person) != y.GetType())){
      throw new ArgumentException("Both objects must be of type Person!");
    }

    return ((Person)x).Age.CompareTo(((Person)y).Age);
  }

  public int Compare(Person x, Person y){
   if((x == null) || (y == null)){
      throw new ArgumentException("Both objects must be of type Person!");
    }

    return x.Age.CompareTo(y.Age);
  }

}
```

Referring to example 11.7 — the PersonAgeComparer is a custom comparer class which can be used to provide a custom ordering of Person objects based on their Age property. (*Note that the "natural" ordering of Person objects is defined by the Person.CompareTo() methods as defined by the Person class.*)

11.8 MainApp.cs

```
using System;
using System.Collections.Generic;

public class MainApp {
  public static void Main(){
    SortedDictionary<PersonKey, Person> _people = new SortedDictionary<PersonKey, Person>();

   Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE, new DateTime(1961, 2, 3),
                          Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE, new DateTime(1972, 1, 1),
                          Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE, new DateTime(1974, 8, 8),
                          Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE, new DateTime(1970, 5, 6),
                          Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
                          new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE, new DateTime(1986, 10, 15),
                          Guid.NewGuid());
```



```

      _people.Add(p1.Key, p1);
      _people.Add(p2.Key, p2);
      _people.Add(p3.Key, p3);
      _people.Add(p4.Key, p4);
      _people.Add(p5.Key, p5);
      _people.Add(p6.Key, p6);


      Console.WriteLine("----- Ordered upon insertion by comparing PersonKeys ------");
      foreach(KeyValuePair<PersonKey, Person> kvp in _people){
        Console.WriteLine(kvp.Value);
      }


      Console.WriteLine("----- Ordered by age with PersonAgeComparer object ------");

      SortedDictionary<Person, Person> _peopleByAge =
                        new SortedDictionary<Person, Person>(new PersonAgeComparer());

      _peopleByAge.Add(p1, p1);
      _peopleByAge.Add(p2, p2);
      _peopleByAge.Add(p3, p3);
      _peopleByAge.Add(p4, p4);
      _peopleByAge.Add(p5, p5);
      _peopleByAge.Add(p6, p6);

      foreach(KeyValuePair<Person, Person> kvp in _peopleByAge){
        Console.WriteLine(kvp.Value);
      }
     }
    }
```

Referring to example 11.8 — The MainApp class given in example 11.8 inserts several Person objects into a SortedDictionary<TKey, TValue> collection and then sorts the items based on their natural ordering. Another instance of SortedDictionary is then created with an instance of PersonAgeComparer being supplied to its constructor. This then allows Person objects to be sorted by their age. Figure 11-14 shows the results of running this program.

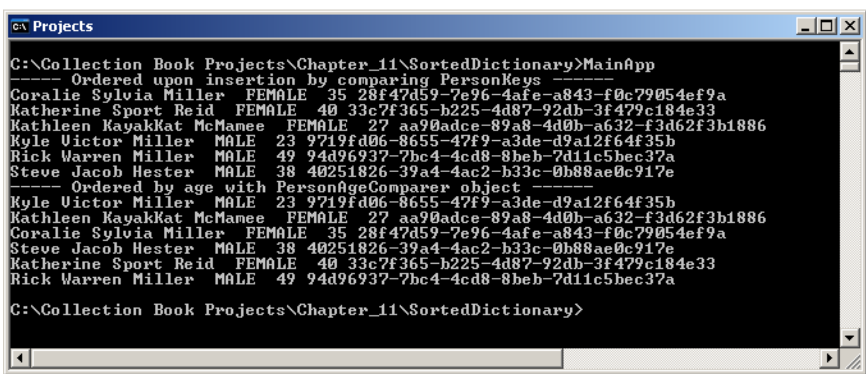


Figure 11-14: Results of Running Example 11.8

Referring to example 11-14 — note that the sorted order of Person objects changed when sorted with the PersonAgeComparer.

Quick Review

The SortedDictionary<TKey, TValue> class uses a red-black tree as its foundational data structure. The SortedDictionary holds items of type KeyValuePair<TKey, TValue>. Items are normally inserted into the collection based on the ordering of supplied keys. You can customize the ordering of items (values) by supplying a custom comparer object. A custom comparer can be created by implementing the IComparer and IComparer<T> interfaces and implementing the appropriate ordering behavior.

SortedList<TKey, TValue>

The SortedList<TKey, TValue> collection stores its items in an array. Like the SortedDictionary collection, the SortedList's items are KeyValuePair<TKey, TValue> objects. However, because SortedList uses internal arrays to store its items, the two classes have different performance characteristics.

Figure 11-15 shows the inheritance hierarchy of the SortedList<TKey, TValue> collection class.

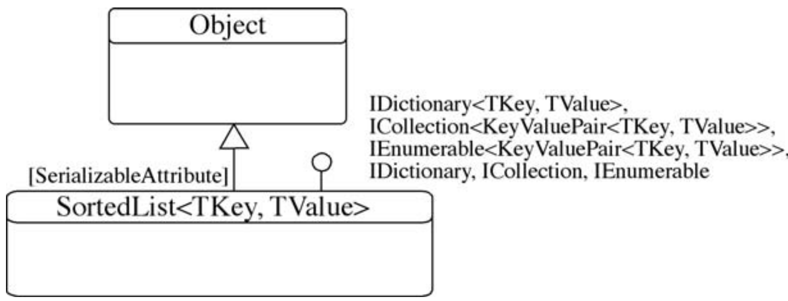


Figure 11-15: SortedList<TKey, TValue> UML Class Diagram

Referring to figure 11-15 — the SortedList<TKey, TValue> collection implements the same interfaces as does the SortedDictionary<TKey, TValue> collection, however, the two classes are not strictly interchangeable because the SortedList class offers several different methods which allow you to get the index of a key or value (i.e., IndexOfKey() and IndexOfValue() methods).

One interface that's not on the list is IList<T>. Thus, even though the SortedList has the word "List" in its name, you cannot directly access an element via its list index like you can with the List<T> class.

Performance Differences Between SortedList and SortedDictionary

The SortedList<TKey, TValue> collection is array-based, while the SortedDictionary<TKey, TValue> collection is red-black tree based. This has some implications on performance you must be aware of, especially if you intend to store large amounts of data in these collections.

As you recall from the red-black tree discussion, when inserting an item into a red-black tree, the incoming item's key is compared with the root node. If it's less than the root node the tree is walked to the left. If it's greater than the root node the tree is walked to the right. These comparisons are made until a null reference is found that can accommodate the incoming item. Thus, the item is sorted in the fly, so to speak. Once the item is placed in the tree, the tree is rebalanced. A balanced tree assures that the time it takes to insert unsorted items happens on average in O(log n) time. — A SortedList will, conversely, perform the item insertion quickly by simply adding it to the end of the array. However, the array must then be sorted, which means that items must be compared until the right spot is found in the array, at which point the item is inserted, and the other items must be shifted to make room for the incoming item. If the items being inserted into a SortedList are already sorted, the SortedList performs better than a SortedDictionary. If you think about this for a moment it makes perfect sense because the SortedDictionary will rebalance the tree even if the items are already in sorted order. On the other hand, if items being inserted into a SortedList are completely unsorted, then all the items on the list must be compared with the incoming item before finding its proper spot in the list. Thus, there is a possibility that an insertion might take O(n) time.

Another performance difference between the two collections is this. When you access the Keys or Values properties of the SortedDictionary<TKey, TValue> class, the tree must be walked each time to generate the requested list. Not so with the SortedList<TKey, TValue> class where the lists returned by the Keys and Values properties are wrappers for the internal keys and values arrays respectively.

SortedList<TKey, TValue> Example

Example 11.9 gives a short program demonstrating the use of the SortedList<TKey, TValue> class.

11.9 SortedListDemo.cs

```
1       using System;
2       using System.Collections.Generic;
3
4       public class SortedListDemo {
5         public static void Main(){
6
7           const string MACBETH_KEY = "MacBeth";
8           const string MACBETH_QUOTE = "I have done the deed. - Didst thou not hear a noise?";
9           const string MACARTHUR_KEY = "MacAuthur";
10          const string MACARTHUR_QUOTE = "Age wrinkles the body. Quitting wrinkles the soul.";
11          const string CHURCHILL_KEY = "Churchill";
12          const string CHURCHILL_QUOTE = "A fanatic is one who can't change his mind and " +
13                                         "won't change the subject.";
14          const string DE_SADE_KEY = "de Sade";
15          const string DE_SADE_QUOTE = "All universal moral principles are idle fancies.";
16
17
18          SortedList<string, string> _quotes = new SortedList<string, string>();
19
20          _quotes.Add(MACBETH_KEY, MACBETH_QUOTE);
21          _quotes.Add(MACARTHUR_KEY, MACARTHUR_QUOTE);
22          _quotes.Add(CHURCHILL_KEY, CHURCHILL_QUOTE);
23          _quotes.Add(DE_SADE_KEY, DE_SADE_QUOTE);
24
25          foreach(KeyValuePair<string, string> kvp in _quotes){
26            Console.WriteLine(kvp.Key + " said:  " + kvp.Value);
27          }
28        }
29      }
```

Referring to example 11.9 — starting on line 7 a series of string keys and values are declared representing several well-known historical figures or characters and their associated quotes. These are then added to the SortedList named _quotes which is declared and created on line 18. The `foreach` statement on line 25 iterates over the collection's KeyValuePair<TKey, TValue> items and prints each key and value to the console. Figure 11-16 shows the results of running this program.

Figure 11-16: Results of Running Example 11.9

Quick Review

The SortedList<TKey, TValue> collection is based on arrays and demonstrates different performance characteristics from the SortedDictionary<TKey, TValue> collection.

Summary

A red-black tree is a special kind of binary tree whose nodes contain an extra piece of information indicating its color which can be either red or black. The red-black tree rebalances itself when necessary after each item insertion by recoloring and rotating nodes based on a set of constraints and cases. The benefit to using a red-black tree is that it won't degrade into a linked-list when fed a list of already-sorted items.

The SortedDictionary<TKey, TValue> class uses a red-black tree as its foundational data structure. The SortedDictionary holds items of type KeyValuePair<TKey, TValue>. Items are normally inserted into the collection based on the ordering of supplied keys. You can customize the ordering of items (values) by supplying a custom comparer object. A custom comparer can be created by implementing the IComparer and IComparer<T> interfaces and implementing the appropriate ordering behavior.

The SortedList<TKey, TValue> collection is based on arrays and demonstrates different performance characteristics from the SortedDictionary<TKey, TValue> collection.

References

Thomas H. Cormen, et. al. *Introduction To Algorithms, Second Edition*. The MIT Press, Cambridge, Massachusetts. ISBN: 0-262-03293-7

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Richard Proudfoot, et. al., editors. *The Arden Shakespeare Complete Works*. Thomas Nelson and Sons Ltd, Walton-on-Thames, Surey, United Kingdom. ISBN: 0-17-4436-15-7

Notes

Chapter 12

Nikon F3HP

USS America (CV-66) Flight Deck

Sets

Learning Objectives

- List and describe the typical operations that can be performed on sets
- State the differences between the HashSet<T> and the SortedSet<T> collections
- List and state the purpose of the interfaces implemented by the HashSet<T> collection
- List and state the purpose of the interfaces implemented by the SortedSet<T> collection
- Explain the operation of the IntersectWith() method
- Explain the operation of the UnionWith() method
- Explain the operation of the SymmetricExceptWith() method
- Explain the operation of the IsSubsetOf() method
- Explain the operation of the IsProperSubsetOf() method
- Explain the operation of the IsSupersetOf() method
- Explain the operation of the IsProperSupersetOf() method
- Explain the operation of the Overlaps() method
- Use the HashSet<T> collection in a program
- Use the SortedSet<T> Collection in a program

Introduction

Many times you'll find it handy to perform basic set operations on a collection. The .NET collections framework provides two collection classes that allow you to do just that. They are the HashSet<T> and the SortedSet<T> classes.

In this chapter I'll introduce you to the HashSet<T> and SortedSet<T> collection classes. I'll explain their differences and demonstrate how to use some of the set manipulation operations they provide. Along the way I will introduce and explain basic set operations as they are supported by these two classes.

HashSet<T> vs. SortedSet<T>

These two classes are similar in that they each store non-duplicate items. The difference between HashSet<T> and SortedSet<T> is that HashSet<T> doesn't impose ordering on its items as does SortedSet<T>. If you want faster performance and don't care in what order the items are stored in the collection, chose HashSet<T>. If you want items to be sorted upon insertion and don't mind the associated performance hit, chose SortedSet<T>.

HashSet<T> Inheritance Hierarchy

Figure 12-1 gives the UML class diagram showing the HashSet<T> inheritance hierarchy.

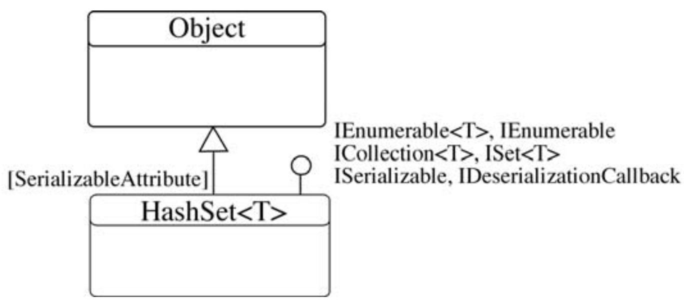


Figure 12-1: HashSet<T> UML Class Diagram

Referring to figure 12-1 — the HashSet<T> collection extends Object and implements the IEnumerable<T>, IEnumerable, ICollection<T>, ISet<T>, ISerializable, and IDeserializationCallback interfaces. The following sections explain the purpose of these interfaces in greater detail.

IEnumerable<T> and IEnumerable

The IEnumerable<T> and IEnumerable interfaces expose an enumerator that is used to iterate over the set using the `foreach` statement. Since the HashSet<T> class imposes no particular order on items in the collection, items extracted via the enumerator are in no particular order. This is especially true after performing set operations that modify the collection such as IntersectWith(), UnionWith(), SymmetricExceptWith(), etc.

ICollection<T>

The ICollection<T> interface extends IEnumerable<T> and IEnumerable and tags the HashSet<T> class as a generic collection. It provides methods such as Add(), Clear(), Contains(), CopyTo(), and Remove(). Note that HashSet<T> does not implement the ICollection interface and instead implements the required methods and properties (i.e., Count, IsSynchronized(), SyncRoot, etc.) directly.

ISet<T>

The ISet<T> interface extends ICollection<T> and exposes the set operation methods: IntersectWith(), UnionWith(), SymmetricExceptWith(), Overlaps(), IsSubsetOf(), IsProperSubsetOf(), IsSupersetOf(), and IsProperSupersetOf().

ISerializable and IDeserializationCallback

The implementation of the ISerializable and IDeserializationCallback interfaces indicate custom serialization.

SortedSet<T> Inheritance Hierarchy

Figure 12-2 gives the UML class diagram showing the SortedSet<T> inheritance hierarchy.

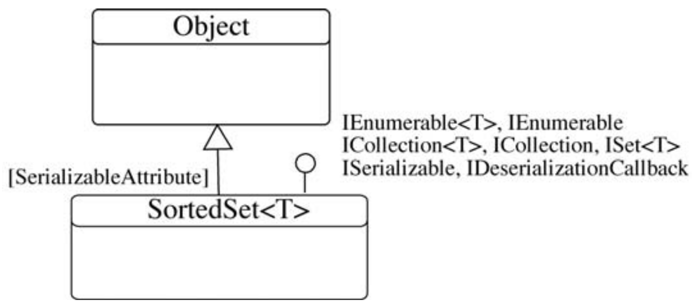


Figure 12-2: SortedSet<T> UML Class Diagram

Referring to figure 12-2 — the SortedSet<T> collection extends Object and implements the IEnumerable<T>, IEnumerable, ICollection<T>, ICollection, ISet<T>, ISerializable, and IDeserializationCallback interfaces. The following sections explain the purpose of these interfaces in greater detail.

IEnumerable<T> and IEnumerable

The IEnumerable<T> and IEnumerable interfaces expose an enumerator that is used to iterate over the set using the `foreach` statement. The items contained within SortedSet<T> are extracted in sorted order.

ICollection<T> and ICollection

The ICollection<T> interface extends IEnumerable<T> and IEnumerable, and ICollection extends IEnumerable. Together these interfaces tag the SortedSet<T> class as a generic collection. They provide methods such as Add(), Clear(), Contains(), CopyTo(), Remove(), IsSynchronized(), and the properties Count and SyncRoot.

ISet<T>

The ISet<T> interface extends ICollection<T> and exposes the set operation methods: IntersectWith(), UnionWith(), SymmetricExceptWith(), Overlaps(), IsSubsetOf(), IsProperSubsetOf(), IsSupersetOf(), and IsProperSupersetOf().

ISerializable and IDeserializationCallback

The implementation of the ISerializable and IDeserializationCallback interfaces indicate custom serialization.

Quick Review

Use HashSet<T> for high performance set operations. It’s faster because it stores its items in no particular order. Use SortedSet<T> when you need set elements stored in sorted order.

Set Operations

Both HashSet<T> and SortedSet<T> provide the same set manipulation operations. These include the following methods: IntersectWith(), UnionWith(), IsProperSubsetOf(), IsProperSupersetOf(), IsSupersetOf(), IsSubsetOf(), Overlaps(), and SymmetricExceptWith(). The following sections explain these operations in greater detail and demonstrate the use of each method.

IntersectWith()

The IntersectWith() method performs an intersection operation on the elements contained in a set and another collection. Figure 12-3 gives the set notation and a Venn diagram for the intersection of two sets A and B.

$$A \cap B = [x | x \in A \wedge x \in B]$$

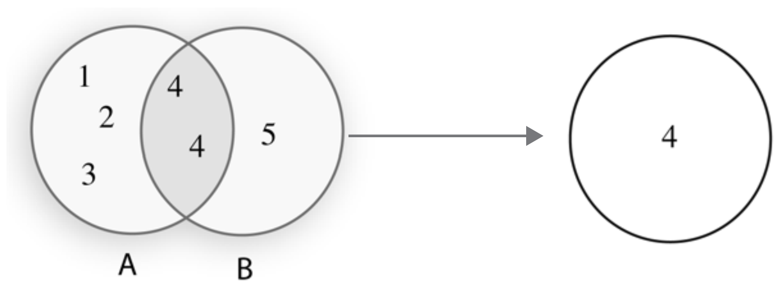


Figure 12-3: Intersection Operation

Referring to figure 12-3 — the set notation at the top of the figure is read: A *intersection* B equals items x where x is a member of A *and* x is a member of B. In the Venn diagram, set A contains the integer items 1, 2, 3, & 4. Set B contains the integer items 4 & 5. The diagram's shaded portion represents the intersection of the two sets. The number 4 is the only item in both A and B and thus the intersection of these two sets results in a new set that contains the number 4. Example 12.1 demonstrates the use of the IntersectWith() method using a HashSet<T> collection.

12.1 IntersectWithDemo.cs

```
using System;
using System.Collections.Generic;

public class IntersectWithDemo {
  public static void Main(){
    HashSet<int> hs = new HashSet<int>();
    List<int> list = new List<int>();

    hs.Add(1);
    hs.Add(2);
    hs.Add(3);
    hs.Add(4);

    list.Add(4);
    list.Add(5);

    Console.WriteLine("Before intersection...");
    Console.Write("The items in HashSet include: ");
    foreach(int i in hs){
      Console.Write(i + " ");
    }
    Console.Write("\nThe items in List include: ");
    foreach(int i in list){
      Console.Write(i + " ");
    }
    Console.WriteLine("\nAfter intersection...");

    hs.IntersectWith(list);
```



```
29
30          Console.Write("The items in HashSet include: ");
31          foreach(int i in hs){
32            Console.Write(i + " ");
33          }
34        }
35      }
```

Referring to example 12.1 — on line 6 a HashSet<int> is declared and created, followed by the creation of a List<int> on the following line. Lines 9 through 12 add the integers 1 through 4 to the HashSet. Lines 14 and 15 add the integers 4 and 5 to the List. The program then writes the contents of each collection to the console before calling the IntersectWith() method on line 28. Note that it's the HashSet that's modified after the call. Figure 12-4 shows the results of running this program.

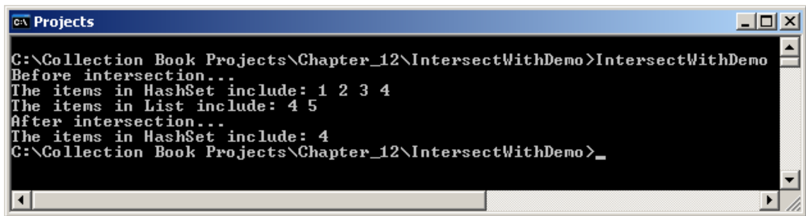


Figure 12-4: Results of Running Example 12.1

UnionWith()

The UnionWith() method performs a union operation on a set and another collection. Figure 12-5 gives the set notation and a Venn diagram for the union of two sets A and B.

$$A \cup B = [x | x \in A \vee x \in B]$$

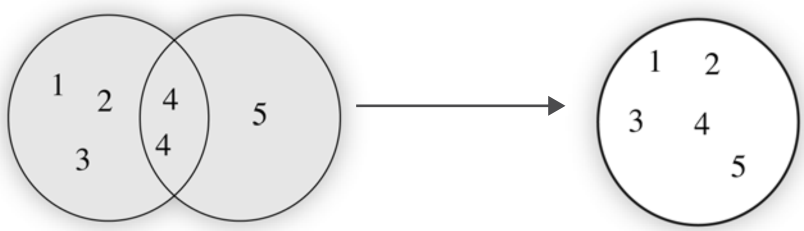


Figure 12-5: Union Operation

Referring to figure 12-5 — the set notation at the top of the diagram is read: A union with B equals x where x is a member of A or x is a member of B. The Venn diagram shows the union of two sets of integers A and B. Set A contains the numbers 1, 2, 3, & 4. Set B contains the numbers 4 & 5. The result of the union is a new set that includes the items contained in both sets.

Example 12.2 demonstrates the use of the UnionWith() method using a HashSet<T> collection.

12.2 UnionWithDemo.cs

```
using System;
using System.Collections.Generic;

public class UnionWithDemo {
  public static void Main(){
    HashSet<int> hs = new HashSet<int>();
    List<int> list = new List<int>();

    hs.Add(1);
    hs.Add(2);
    hs.Add(3);
    hs.Add(4);

    list.Add(4);
    list.Add(5);
```



```
16
17          Console.WriteLine("Before union...");
18          Console.Write("The items in HashSet include: ");
19          foreach(int i in hs){
20            Console.Write(i + " ");
21          }
22          Console.Write("\nThe items in List include: ");
23          foreach(int i in list){
24            Console.Write(i + " ");
25          }
26          Console.WriteLine("\nAfter union...");
27
28          hs.UnionWith(list);
29
30          Console.Write("The items in HashSet include: ");
31          foreach(int i in hs){
32            Console.Write(i + " ");
33          }
34        }
35      }
```

Referring to example 12.2 — this example is similar to the IntersectWithDemo code except that the console outputs have been modified to reflect the union operation and the UnionWith() method is called on line 28. Figure 12-6 shows the results of running this program.

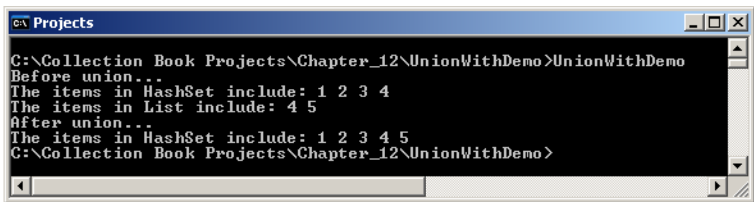


Figure 12-6: Results of Running Example 12.2

IsSubsetOf()

The IsSubsetOf() method compares the contents of a set with the contents of another collection and returns true if the comparing set is a subset of the compared collection. A set is a subset of itself and an empty set is a subset of any set. Figure 12-7 shows the set notation and the Venn diagram for the subset relationship.

$$A \subseteq B = [x | \forall x \in A \text{ and } x \in B]$$

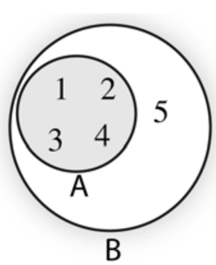


Figure 12-7: Subset Relationship

Referring to figure 12-7 — the set notation at the top of the figure is read: A is a subset of B is x such that for all x in A, x is also in B. The Venn diagram shows two sets: A = {1, 2, 3, 4} and B = {1, 2, 3, 4, 5}. A is a subset of B because all of the elements of A are also contained in B. The set A could also be equal to B and still be a subset.

Example 12.3 demonstrates the use of the IsSubsetOf() method.

12.3 IsSubsetOfDemo.cs

```
using System;
using System.Collections.Generic;

public class IsSubsetOfDemo {
  public static void Main(){

    HashSet<int> emptySet = new HashSet<int>();

    HashSet<int> A = new HashSet<int>();
    A.Add(1);
    A.Add(2);
    A.Add(3);
    A.Add(4);

    HashSet<int> B = new HashSet<int>();
    B.Add(1);
    B.Add(2);
    B.Add(3);
    B.Add(4);
    B.Add(5);

    Console.Write("Empty Set Contents: ");
    foreach(int i in emptySet){
      Console.Write(i + " ");
    }

    Console.Write("\nSet A Contents: ");
    foreach(int i in A){
      Console.Write(i + " ");
    }

    Console.Write("\nSet B Contents: ");
    foreach(int i in B){
      Console.Write(i + " ");
    }

    Console.WriteLine("\n");

    Console.WriteLine("A.IsSubsetOf(B) = " + A.IsSubsetOf(B));
    Console.WriteLine("emptySet.IsSubsetOf(A) = " + emptySet.IsSubsetOf(A));
    Console.WriteLine("emptySet.IsSubsetOf(B) = " + emptySet.IsSubsetOf(B));
    Console.WriteLine("B.IsSubsetOf(A) = " + B.IsSubsetOf(A));

    Console.WriteLine("\nAdding the number 5 to set A...");
    A.Add(5);

    Console.Write("\nSet A Contents: ");
    foreach(int i in A){
      Console.Write(i + " ");
    }

    Console.Write("\nSet B Contents: ");
    foreach(int i in B){
      Console.Write(i + " ");
    }

    Console.WriteLine("\nA.IsSubsetOf(B) = " + A.IsSubsetOf(B));

    Console.WriteLine("\nAdding the number 6 to set A...");
    A.Add(6);

    Console.Write("\nSet A Contents: ");
    foreach(int i in A){
      Console.Write(i + " ");
    }

    Console.Write("\nSet B Contents: ");
    foreach(int i in B){
      Console.Write(i + " ");
    }

    Console.WriteLine("\nA.IsSubsetOf(B) = " + A.IsSubsetOf(B));

  }
}
```

Referring to example 12.3 — three HashSet<int> references are declared and initialized named: emptySet, A, and B. The elements {1, 2, 3, 4} are added to set A and the elements {1, 2, 3, 4, 5} are added to set B. The emptySet is left empty. The program then prints the contents of each collection to the console before calling the IsSubsetOf()

method via the various references. On line 45 the number 5 is added to set A and a comparison between sets A and B is made. The number 6 is then added to set A on line 60 and a final comparison is made between sets A and B. Figure 12-8 shows the results of running this program.

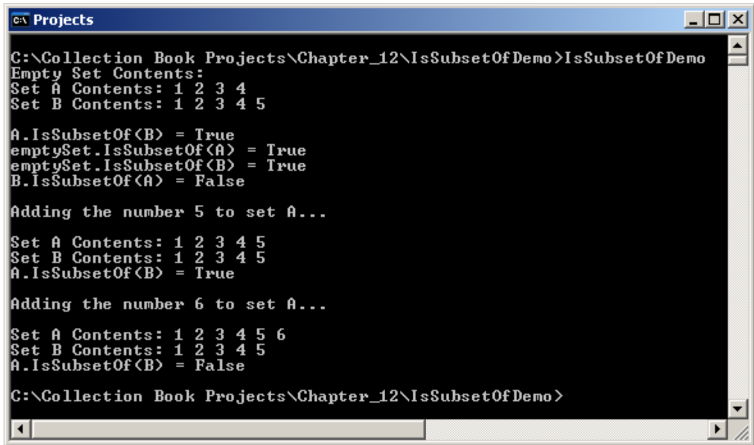


Figure 12-8: Results of Running Example 12.3

IsProperSubsetOf()

The IsProperSubsetOf() method compares the contents of a set with the contents of another collection and returns true if the set is a subset of the compared collection but does not equal the compared collection. In other words, if the set A = {1, 2, 3, 4} and set B = {1, 2, 3, 4, 5} then set A is both a subset of B and a proper subset of B because A is not equal to B. Figure 12-9 gives the set notation and a Venn diagram for the proper subset relationship between two sets A and B.

$$A \subset B = [x | \forall x \in A \text{ and } x \in B \text{ and } \mathrm{A} \neq B]$$

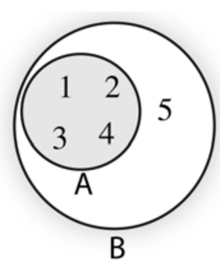


Figure 12-9: Proper Subset Relationship

Referring to figure 12-9 — the set notation at the top of the figure is read: A proper subset B is x such that for all x in A, x is also in B, and A is not equal to B.

Example 12.4 demonstrates the use of the IsProperSubset() method.

12.4 IsProperSubsetOfDemo.cs

```
using System;
using System.Collections.Generic;

public class IsProperSubsetOfDemo {
  public static void Main(){

    HashSet<int> emptySet = new HashSet<int>();

    HashSet<int> A = new HashSet<int>();
    A.Add(1);
    A.Add(2);
    A.Add(3);
    A.Add(4);
```



```
        HashSet<int> B = new HashSet<int>();
        B.Add(1);
        B.Add(2);
        B.Add(3);
        B.Add(4);
        B.Add(5);

        Console.Write("Empty Set Contents: ");
        foreach(int i in emptySet){
          Console.Write(i + " ");
        }

        Console.Write("\nSet A Contents: ");
        foreach(int i in A){
          Console.Write(i + " ");
        }

        Console.Write("\nSet B Contents: ");
        foreach(int i in B){
          Console.Write(i + " ");
        }

        Console.WriteLine("\n");

        Console.WriteLine("A.IsProperSubsetOf(B) = " + A.IsProperSubsetOf(B));
        Console.WriteLine("emptySet.IsProperSubsetOf(A) = " + emptySet.IsProperSubsetOf(A));
        Console.WriteLine("emptySet.IsProperSubsetOf(B) = " + emptySet.IsProperSubsetOf(B));
        Console.WriteLine("B.IsproperSubsetOf(A) = " + B.IsProperSubsetOf(A));

        Console.WriteLine("\nAdding the number 5 to set A...");
        A.Add(5);

        Console.Write("\nSet A Contents: ");
        foreach(int i in A){
          Console.Write(i + " ");
        }

        Console.Write("\nSet B Contents: ");
        foreach(int i in B){
          Console.Write(i + " ");
        }

        Console.WriteLine("\nA.IsSubsetOf(B) = " + A.IsSubsetOf(B));
        Console.WriteLine("A.IsProperSubsetOf(B) = " + A.IsProperSubsetOf(B));

        Console.WriteLine("\nAdding the number 6 to set A...");
        A.Add(6);

        Console.Write("\nSet A Contents: ");
        foreach(int i in A){
          Console.Write(i + " ");
        }

        Console.Write("\nSet B Contents: ");
        foreach(int i in B){
          Console.Write(i + " ");
        }

        Console.WriteLine("\nA.IsSubsetOf(B) = " + A.IsSubsetOf(B));
        Console.WriteLine("A.IsProperSubsetOf(B) = " + A.IsProperSubsetOf(B));
        Console.WriteLine("B.IsSubsetOf(A) = " + B.IsSubsetOf(A));
        Console.WriteLine("B.IsProperSubsetOf(A) = " + B.IsProperSubsetOf(A));
      }
    }
```

Referring to example 12.4 — this program expands on the previous example but compares the sets using the IsProperSubsetOf() method. Figure 12-10 shows the results of running this program.

IsSupersetOf()

The IsSupersetOf() method compares the elements of a set with the elements of another collection and returns true if the comparing set is a superset of the compared collection. A set A is a superset of another set B if all the elements in set B are also contained in set A. This is another way of saying that set B is a subset of set A. A set is a superset of itself and two equal sets are supersets of each other. Figure 12-11 gives the set notation and Venn diagram for the superset relationship.

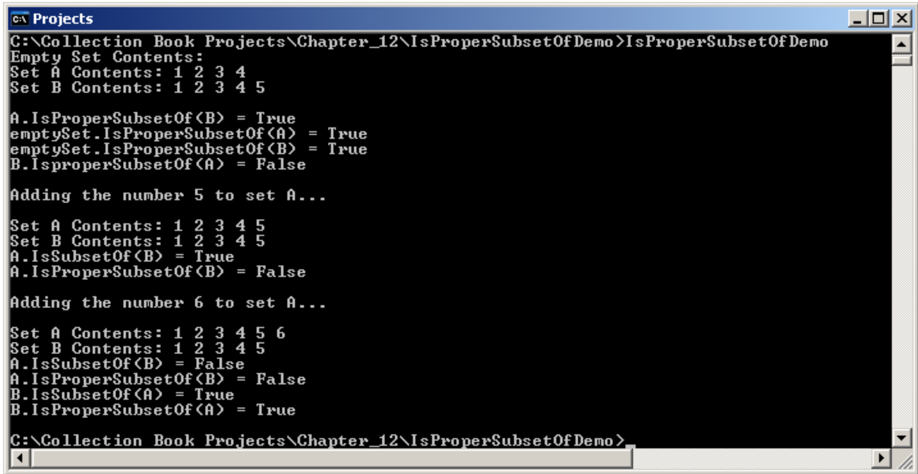


Figure 12-10: Results of Running Example 12.4

$$A \supseteq B \;=\; [x | \forall x \in \; B \text{ and } \; x \in \; A]$$

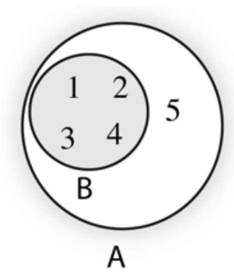


Figure 12-11: Superset Relationship

Referring to figure 12-11 — the set notation in the upper part of the figure is read: A super set B is x such that for all x that are members of B, x is also a member of A. The Venn diagram shows two sets: set A with elements {1, 2, 3, 4, 5} and set B with elements {1, 2, 3, 4}. Set A is a superset of B, and conversly, set B is a subset of A. In this diagram, set B is also a proper subset of set A.

Example 12.5 demonstrates the use of the IsSupersetOf() method.

12.5 IsSupersetOfDemo.cs

```
using System;
using System.Collections.Generic;

public class IsSupersetOfDemo {
  public static void Main(){

    HashSet<int> emptySet = new HashSet<int>();

    HashSet<int> A = new HashSet<int>();
    A.Add(1);
    A.Add(2);
    A.Add(3);
    A.Add(4);

    HashSet<int> B = new HashSet<int>();
    B.Add(1);
    B.Add(2);
    B.Add(3);
    B.Add(4);
    B.Add(5);

    Console.Write("Empty Set Contents: ");
    foreach(int i in emptySet){
```



```
24              Console.Write(i + " ");
25            }
26
27            Console.Write("\nSet A Contents: ");
28            foreach(int i in A){
29              Console.Write(i + " ");
30            }
31
32            Console.Write("\nSet B Contents: ");
33            foreach(int i in B){
34              Console.Write(i + " ");
35            }
36
37            Console.WriteLine("\n");
38
39            Console.WriteLine("A.IsSupersetOf(B) = " + A.IsSupersetOf(B));
40            Console.WriteLine("emptySet.IsSupersetOf(A) = " + emptySet.IsSupersetOf(A));
41            Console.WriteLine("emptySet.IsSupersetOf(B) = " + emptySet.IsSupersetOf(B));
42            Console.WriteLine("B.IsSupersetOf(A) = " + B.IsSupersetOf(A));
43
44            Console.WriteLine("\nAdding the number 5 to set A...");
45            A.Add(5);
46
47            Console.Write("\nSet A Contents: ");
48            foreach(int i in A){
49              Console.Write(i + " ");
50            }
51
52            Console.Write("\nSet B Contents: ");
53            foreach(int i in B){
54              Console.Write(i + " ");
55            }
56
57            Console.WriteLine("\nA.IsSupersetOf(B) = " + A.IsSupersetOf(B));
58            Console.WriteLine("A.IsSupersetOf(A) = " + A.IsSupersetOf(A));
59
60            Console.WriteLine("\nAdding the number 6 to set A...");
61            A.Add(6);
62
63            Console.Write("\nSet A Contents: ");
64            foreach(int i in A){
65              Console.Write(i + " ");
66            }
67
68            Console.Write("\nSet B Contents: ");
69            foreach(int i in B){
70              Console.Write(i + " ");
71            }
72
73            Console.WriteLine("\nA.IsSupersetOf(B) = " + A.IsSupersetOf(B));
74          }
75        }
```

Referring to example 12.5 — three HashSet<int> references named emptySet, A, and B are declared and initialized. The elements {1, 2, 3, 4} are added to set A and the elements {1, 2, 3, 4, 5} are added to set B. The emptySet is left empty. The program then prints the contents of each set to the console before making a series of set comparisons using the IsSupersetOf() method on lines 39 through 42. Next, the number 5 is added to set A. The contents of sets A and B are again written to the console followed by the comparison of set A to B and of set A with itself. On line 61 the number 6 is added to set A and again the contents of each set A and B is written to the console. Sets A is compared with set B one final time on line 73. Figure 12-12 shows the results of running this program.

IsProperSupersetOf()

The IsProperSupersetOf() method compares the contents of a set with the contents of another collection and returns true if the set is a proper superset of the compared collection. A proper superset is different from a superset in that a proper superset must contain an additional element not found in the contained subset. If set A = {1, 2, 3, 4, 5} and set B = {1, 2, 3, 4} then set A is a proper superset of set B. Set B is a subset of set A and is also a proper subset. Figure 12-13 shows the set notation and Venn diagram for the proper superset relationship.

Referring to figure 12-13 — the set notation at the top of the figure is read: A proper superset B is x such that for all x in B, x is also a member of A, and A is not equal to B. The Venn diagram shows two sets: set A = {1, 2, 3, 4, 5} and set B = {1, 2, 3, 4}. Set A is a proper superset of set B because set A contains all the elements of set B plus one additional element. Example 12.6 demonstrates the use of the IsProperSupersetOf() method.

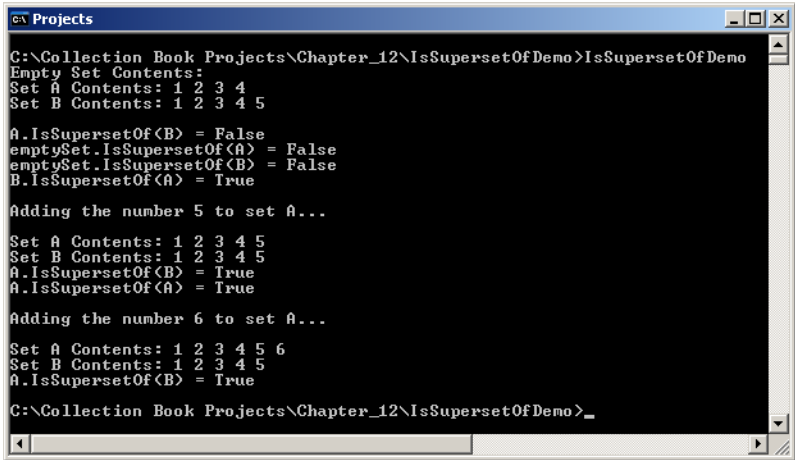


Figure 12-12: Results of Running Example 12.5

$$A \supset B = [x | \forall x \in B \text{ and } x \in A \text{ and } A \neq B]$$

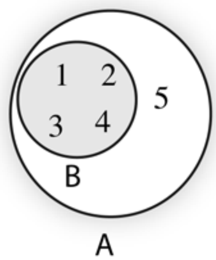


Figure 12-13: Proper Superset Relationship

12.6 IsProperSupersetOfDemo.cs

```
using System;
using System.Collections.Generic;

public class IsProperSupersetOfDemo {
  public static void Main(){

    HashSet<int> emptySet = new HashSet<int>();

    HashSet<int> A = new HashSet<int>();
    A.Add(1);
    A.Add(2);
    A.Add(3);
    A.Add(4);

    HashSet<int> B = new HashSet<int>();
    B.Add(1);
    B.Add(2);
    B.Add(3);
    B.Add(4);
    B.Add(5);

    Console.Write("Empty Set Contents: ");
    foreach(int i in emptySet){
      Console.Write(i + " ");
    }

    Console.Write("\nSet A Contents: ");
    foreach(int i in A){
      Console.Write(i + " ");
    }

    Console.Write("\nSet B Contents: ");
```



```
33         foreach(int i in B){
34           Console.Write(i + " ");
35         }
36
37         Console.WriteLine("\n");
38
39         Console.WriteLine("A.IsProperSupersetOf(B) = " + A.IsProperSupersetOf(B));
40         Console.WriteLine("emptySet.IsProperSupersetOf(A) = " + emptySet.IsProperSupersetOf(A));
41         Console.WriteLine("emptySet.IsProperSupersetOf(B) = " + emptySet.IsProperSupersetOf(B));
42         Console.WriteLine("B.IsproperSupersetOf(A) = " + B.IsProperSupersetOf(A));
43
44         Console.WriteLine("\nAdding the number 5 to set A...");
45         A.Add(5);
46
47         Console.Write("\nSet A Contents: ");
48         foreach(int i in A){
49           Console.Write(i + " ");
50         }
51
52         Console.Write("\nSet B Contents: ");
53         foreach(int i in B){
54           Console.Write(i + " ");
55         }
56
57         Console.WriteLine("\nA.IsSupersetOf(B) = " + A.IsSupersetOf(B));
58         Console.WriteLine("A.IsProperSupersetOf(B) = " + A.IsProperSupersetOf(B));
59
60         Console.WriteLine("\nAdding the number 6 to set A...");
61         A.Add(6);
62
63         Console.Write("\nSet A Contents: ");
64         foreach(int i in A){
65           Console.Write(i + " ");
66         }
67
68         Console.Write("\nSet B Contents: ");
69         foreach(int i in B){
70           Console.Write(i + " ");
71         }
72
73         Console.WriteLine("\nA.IsSupersetOf(B) = " + A.IsSupersetOf(B));
74         Console.WriteLine("A.IsProperSupersetOf(B) = " + A.IsProperSupersetOf(B));
75         Console.WriteLine("B.IsSupersetOf(A) = " + B.IsSupersetOf(A));
76         Console.WriteLine("B.IsProperSupersetOf(A) = " + B.IsProperSupersetOf(A));
77       }
78     }
```

Referring to example 12.6 — three HashSet<int> references named emptySet, A, and B are declared and initialized. Sets A and B are populated with elements and the emptySet is left empty. Their contents are printed to the console and then the IsProperSupersetOf() method is used to compare the sets to each other and the results are printed to the console. On line 45 the number 5 is added to set A and again the contents of sets A and B are printed to the console followed by the comparison of set A with set B. On line 61 the number 6 is added to set A and the contents of both sets are again printed to the console followed by another set of comparisons between sets A and B using the IsSupersetOf() and IsProperSupersetOf() methods. Figure 12-14 shows the results of running this program.

Overlaps()

The Overlaps() method compares the elements of a set with the elements of another collection and returns true if the set contains elements in common with the compared collection. It's equivalent to performing a non-mutating intersection since the result of the method call is simply a boolean value that's true if there are elements in common and false otherwise. Figure 12-15 shows the set notation and Venn diagram for the intersection operation performed by the Overlaps() method.

Referring to figure 12-15 — the set notation at the top of the figure reads: A union B is x such that x is a member of set A and x is a member of set B. The Venn diagram illustrates this concept with set A = {1, 2, 3, 4} and set B = {1, 2, 3, 4, 5}. Example 12.7 demonstrates the use of the Overlaps() method.

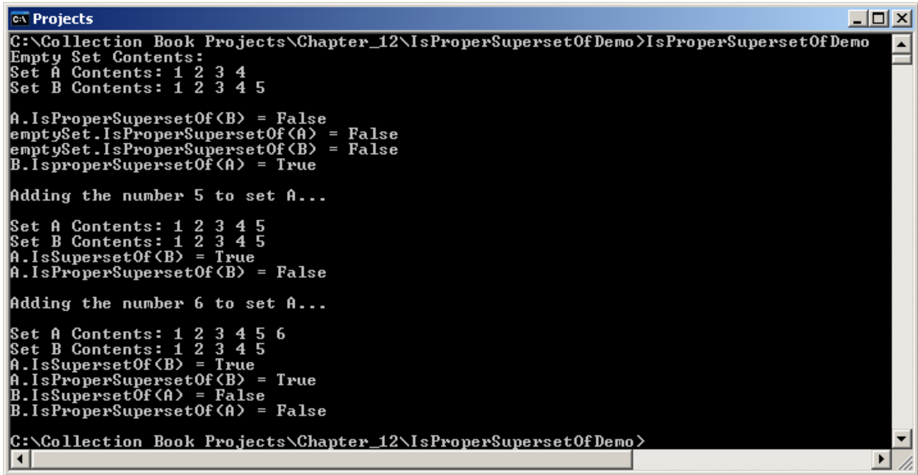


Figure 12-14: Results of Running Example 12.6

$$A \cap B = [x | x \in A \wedge x \in B]$$

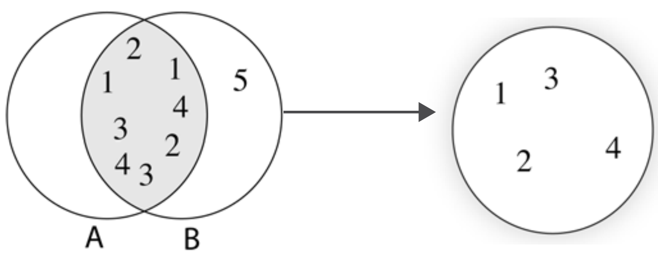


Figure 12-15: Overlap Looks for Elements Common to Each Set

12.7 OverlapsDemo.cs

```
using System;
using System.Collections.Generic;

public class OverlapsDemo {
  public static void Main(){

    HashSet<int> emptySet = new HashSet<int>();

    HashSet<int> A = new HashSet<int>();
    A.Add(1);
    A.Add(2);
    A.Add(3);
    A.Add(4);

    HashSet<int> B = new HashSet<int>();
    B.Add(1);
    B.Add(2);
    B.Add(3);
    B.Add(4);
    B.Add(5);

    Console.Write("Empty Set Contents: ");
    foreach(int i in emptySet){
      Console.Write(i + " ");
    }

    Console.Write("\nSet A Contents: ");
    foreach(int i in A){
      Console.Write(i + " ");
    }
```

```
31
32        Console.Write("\nSet B Contents: ");
33        foreach(int i in B){
34          Console.Write(i + " ");
35        }
36
37        Console.WriteLine("\n");
38
39        Console.WriteLine("A.Overlaps(B) = " + A.Overlaps(B));
40        Console.WriteLine("emptySet.Overlaps(A) = " + emptySet.Overlaps(A));
41        Console.WriteLine("emptySet.Overlaps(B) = " + emptySet.Overlaps(B));
42        Console.WriteLine("B.Overlaps(A) = " + B.Overlaps(A));
43
44        Console.WriteLine("\nAdding the number 5 to set A...");
45        A.Add(5);
46
47        Console.Write("\nSet A Contents: ");
48        foreach(int i in A){
49          Console.Write(i + " ");
50        }
51
52        Console.Write("\nSet B Contents: ");
53        foreach(int i in B){
54          Console.Write(i + " ");
55        }
56
57        Console.WriteLine("\nA.Overlaps(B) = " + A.Overlaps(B));
58
59        Console.WriteLine("\nAdding the number 6 to set A...");
60        A.Add(6);
61
62        Console.Write("\nSet A Contents: ");
63        foreach(int i in A){
64          Console.Write(i + " ");
65        }
66
67        Console.Write("\nSet B Contents: ");
68        foreach(int i in B){
69          Console.Write(i + " ");
70        }
71
72        Console.WriteLine("\nA.Overlaps(B) = " + A.Overlaps(B));
73      }
74    }
```

Referring to example 12.7 — three HashSet<int> sets are declared and initialized: emptySet, A, and B. The items {1, 2, 3, 4} are added to set A, and the items {1, 2, 3, 4, 5} are added to set B. The emptySet is left empty. The contents of each set is written to the console followed by calls to the Overlaps() method to compare each set. On line 45 the number 5 is added to set A, the contents of sets A and B are written to the console, and the Overlaps() method is called again on line 57. On line 60 the number 6 is added to set A and the comparison is repeated. Figure 12-16 shows the results of running this program

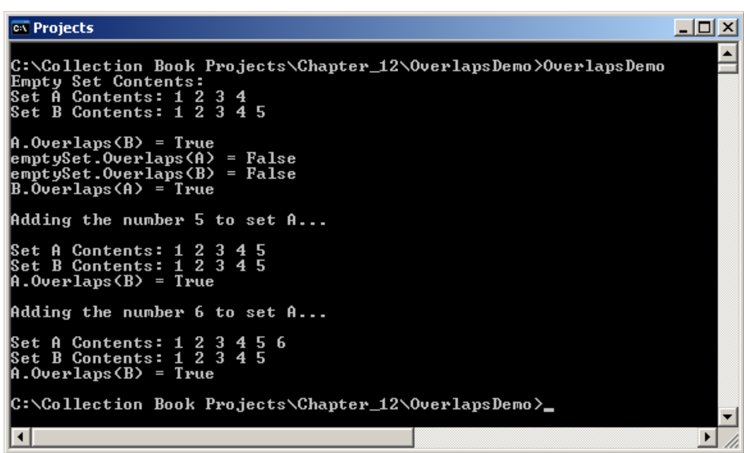


Figure 12-16: Results of Running Example 12.7

SymmetricExceptWith()

The SymmetricExceptWith() method performs a symmetric difference operation on a set and another collection with the result being the unique items that appear in each collection, but not both collections. Figure 12-17 gives the set notation and Venn diagram for the symmetric difference operation.

$$A \;\Delta\; B \;=\; [x|(x \in \; A \wedge x \notin \; B) \text{ or } (x \in \; B \wedge x \notin \; A)]$$

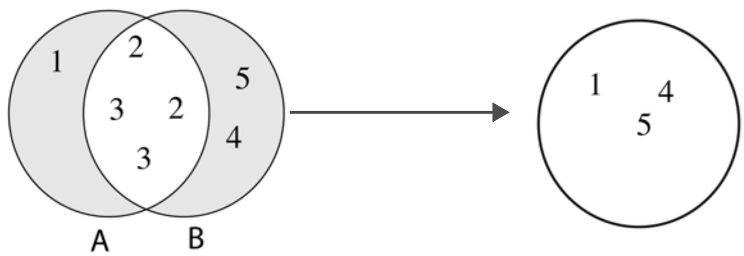


Figure 12-17: Symmetric Difference

Referring to figure 12-17 — the set notation at the top of the figure is read: A symmetric difference B is x such that x is a member of set A but not a member of set B or x is a member of set B but not a member of set A. The Venn diagrams show two sets A and B. Set A = {1, 2, 3} and set B = {2, 3, 4, 5}. The symmetric difference between these two sets is the set that contains the elements {1, 4, 5}. These are the elements that appear in either set A or set B but not both. Example 12.8 demonstrates the use of the SymmetricExceptWith() method.

12.8 SymmetricExceptWithDemo.cs

```
1     using System;
2     using System.Collections.Generic;
3
4     public class SymmetricExceptWithDemo {
5       public static void Main(){
6         HashSet<int> A = new HashSet<int>();
7         A.Add(1);
8         A.Add(2);
9         A.Add(3);
10
11        HashSet<int> B = new HashSet<int>();
12        B.Add(2);
13        B.Add(3);
14        B.Add(4);
15        B.Add(5);
16
17        Console.Write("Contents of set A: ");
18        foreach(int i in A){
19          Console.Write(i + " ");
20        }
21
22        Console.Write("\nContents of set B: ");
23        foreach(int i in B){
24          Console.Write(i + " ");
25        }
26
27        A.SymmetricExceptWith(B);
28
29        Console.Write("\nContents of A after A.SymmetricExceptWith(B) = ");
30        foreach(int i in A){
31          Console.Write(i + " ");
32        }
33      }
34    }
```

Referring to example 12.8 — two HastSet<int> references named A and B are declared and initialized. To set A are added the items {1, 2, 3} and to set B the items {2, 3, 4, 5}. The contents of both sets A and B are then written to the console. On line 27 the SymmetricExceptWith() method is called via set A passing set B in as an argument. This results in set A being modified to contain the symmetric difference between both sets. The program ends by writing the contents of set A to the console. Figure 12-18 shows the results of running this program.

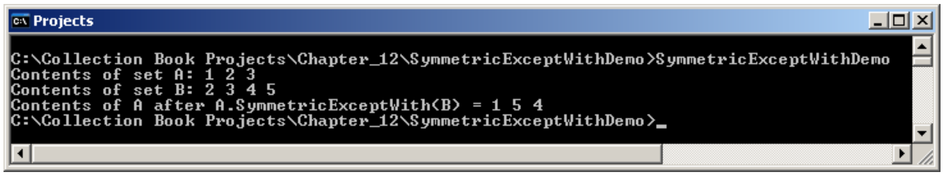


Figure 12-18: Results of Running Example 12.8

Quick Review

The set manipulation operations include: IntersectionWith(), UnionWith(), Overlaps(), SymmetricExceptWith(), IsSubsetOf(), IsProperSubsetOf(), IsSupersetOf(), and IsProperSupersetOf().

Summary

The set collections include HashSet<T> and SortedSet<T>. Sets cannot contain duplicate items.

Use HashSet<T> for high performance set operations. It's faster because it stores its items in no particular order. Use SortedSet<T> when you need set elements stored in sorted order.

The set manipulation operations include: IntersectionWith(), UnionWith(), Overlaps(), SymmetricExceptWith(), IsSubsetOf(), IsProperSubsetOf(), IsSupersetOf(), and IsProperSupersetOf().

References

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Thomas H. Cormen, et. al. *Introduction To Algorithms, Second Edition*. The MIT Press, Cambridge, Massachusetts. ISBN: 0-262-03293-7

Notes

Chapter 13

Nikon F3HP

USS America (CV-66) Flight Deck

Thread Programming

Learning Objectives

- Define the term "thread safe"
- Create managed threads using the Thread class
- List and describe the managed thread states
- Start a thread with a ThreadStartDelegate
- Start a thread with a ParameterizedThreadStartDelegate
- Pass an argument to a thread using a ParameterizedThreadStartDelegate
- Describe the difference between foreground threads and background threads
- Use the ThreadPool class to tap into a thread pool
- Create and use delegates to call methods asychronously
- Use thread synchronization as it applies to collections

Introduction

When you look at the MSDN documentation for collection classes you'll read passages that refer to certain methods as being thread safe or not. The purpose of this chapter is to introduce you to the subject of threads and multithreaded programming to prepare you for the following chapter on threads and collections. While the term *multithreaded programming* may sound complicated, it is in reality quite easy to do in C# .NET.

In this chapter I will explain how multithreading works on your typical, general-purpose computer. You'll learn about the relationship between a process and its threads and how an operating system manages thread execution. I'll then show you how to use the Thread class in your programs to create and start managed threads. Next, I'll show you how you can simplify the creation and management of multiple threads with the help of the BackgroundWorker class. Also, I'll show you how to use ThreadPool threads and how to run any method asynchronously with the help of delegates.

As is the case with any tool, there's a right way to use it and a wrong way. Multithreading, applied thoughtlessly, will render your programs overly complicated, sluggish, unresponsive, and buggy. But, when used with care, multithreading can significantly increase your application's performance, giving it that hard-to-describe-but-you-know-it-when-you-see-it feeling of professionalism.

Before getting started I need to add a caveat. While I present a lot of material in this chapter, I make no attempt to cover all aspects of multithreaded programming. To do so would bore you to death, and in fact, as it turns out, a lot of what you can do with threads you shouldn't do. Instead, I will focus on those topics that give you a lot of bang for your buck to get you up and running as quickly as possible with multithreaded programming. In some cases, I have postponed the discussion of more obscure threading topics until later in the book where their presentation is more appropriate.

As usual, I recommend that if you want to dive deeper into threads and multithreaded programming consult one of the excellent references I've listed at the end of the chapter.

Multithreading Overview: The Tale Of Two Vacations

As I write this, winter is is full swing in Northern Virginia. I can hardly wait for summer to arrive so I can go on vacation. Let's take a look at the concept of vacation from a thread's point of view.

Single-Threaded Vacation

Imagine for a moment you're on vacation, trying to relax on the squeaky white sand of a sun-drenched tropical beach. Your job, since you are on vacation, is to relax, and as you start to drift off for a snooze you get thirsty. You are the only one on the beach and the bar is a mile away! You get up and walk to the bar and buy a drink, no, better make that two drinks, and walk back to your lounge chair. Now you start to relax again, until you get hungry. The grill is a mile in the other direction, so you get up again and walk to the grill. What you really want to do is relax and enjoy the beach, but what you ended up doing was a little relaxing, some drink fetching, and some food hunting. Eventually you'll get back to relaxing. After all, you're on single-threaded vacation.

Multithreaded Vacation

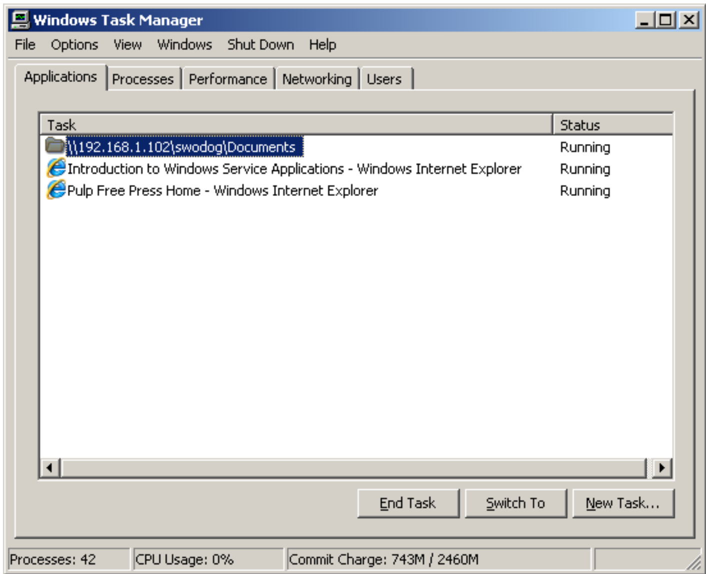
Now imagine that you're on multithreaded vacation. Again, your job is to relax on the beach. This time, however, when you get thirsty, you ask the wait staff to please bring you a drink, which they immediately set out to do, while you immediately return to relaxing. When you get hungry, you again summon the wait staff and off they go to fetch you a little somethin' somethin' from the grill. You immediately return to relaxing. Multithreaded vacation is so much better! If you've ever returned from a vacation and felt like you needed a vacation, you probably didn't take a multithreaded vacation because you were never allowed to relax!

The Relationship Between A Process And Its Threads

In the tale of two vacations above, you could think of yourself as being a process: the "Relax" process. On a computer, the operating system loads and starts services and applications. Each service or application runs as a separate *process* on the machine. (**Note:** A *service* is a special type of Windows application that runs solely in the background with limited or no user interaction.) Figure 13-1 shows a list of applications running on my machine as I write these words. Figure 13-2 shows a list of processes.

3 applications

42 processes



Click CTRL-ALT-DELETE to display the Windows Task Manager window.

Figure 13-1: List of Running Applications

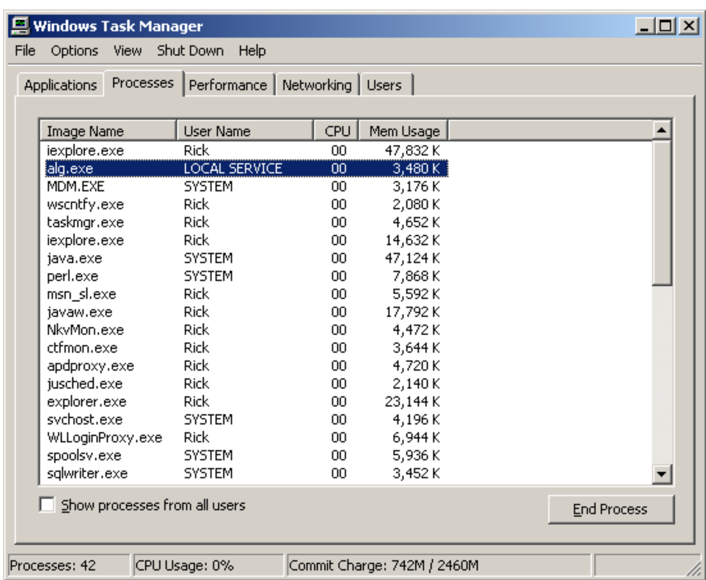


Figure 13-2: Partial List of Processes Running on the Same Computer

Referring to figures 13-1 and 13-2 — there are quite a few more processes actually running than there are applications. Many of the processes are background operating system processes that are started automatically when the computer powers up. There are two databases running: MS/SQL Server Express and Oracle 10G. You can also see in the process list that Java (java.exe) and a Perl interpreter (perl.exe) are running along with the Windows Desktop Explorer (explorer.exe) and two instances of Internet Explorer (iexplore.exe). Each one of these processes is isolated

from the others meaning that each process has an allocated memory space all to itself. The management of this process memory space is left to the operating system.

A process consists of one or more *threads of execution*, referred to simply as *threads*. A process always consists of at least one thread, the *Main thread*, (the Main() method's thread of execution) which starts running when the process begins execution. A *single-threaded process* contains only one thread of execution. A *multithreaded process* contains more than one thread. Figure 13-3 offers a representation of processes and threads in a single-processor system.

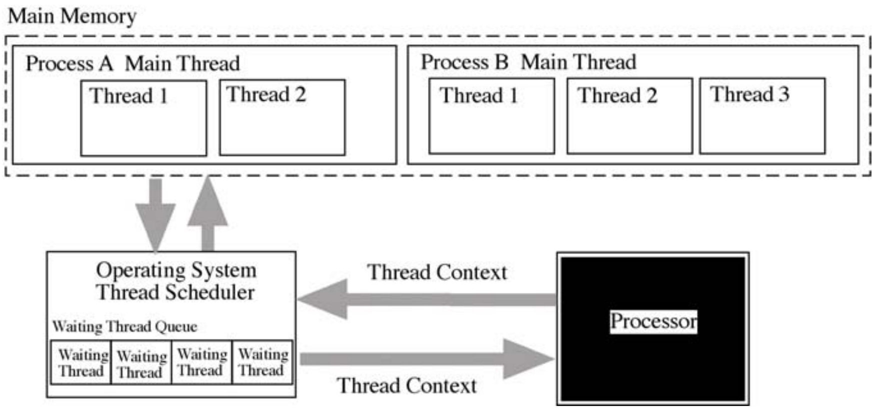


Figure 13-3: Processes and their Threads Executing in a Single-Processor Environment

Referring to figure 13-3 — two processes A and B are executing. Process A contains three threads: Main, Thread 1 and Thread 2. Process B contains four threads: Main, Thread 1, Thread 2, and Thread 3. A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and *application domain*.

As you can see in figure 13-3, the operating system thread scheduler coordinates thread execution. Waiting threads sit in a *thread queue* until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system, the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor. This diagram makes clear that in a single-processor system, the notion of concurrently executing applications is just an illusion pulled off by the operating system quickly switching threads in and out of the processor. Figure 13-4 shows how things might look on a multiprocessor system. Referring to figure 13-4 — now we can really get some work done. In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Returning once again to my earlier vacation analogy, when you're on single-threaded vacation, the relax process does everything related to the vacation in one thread of execution. That's why you must stop relaxing and fetch yourself a drink and something to eat. When you're on multithreaded vacation, the relax process concentrates on relaxing and hands off the chores of drink and food fetching to separate threads, You come away from a multithreaded vacation feeling a lot more relaxed! (*Well, at least until you arrive at the airport anyway.*)

Vacation Gone Bad

There is a possibility, even on multithreaded vacation, for you to return home tense and frustrated. This can occur if the drink and food fetching threads misbehave. How might this happen? Assume for a moment, if you will, that you are not the only process on the beach. Laying next to you is a nasty little someone named "create-hate-and-discontent". He told his food and drink fetching threads they were special and gave them an order to cut in front of the line whenever possible. Your threads get booted from the bar and grill counters more frequently because of the higher priority of create-hate-and-discontent's threads. You suffer because your threads take longer to fetch food and drink. This is only one example of how ill-behaved threads can bring one or more processes to a halt.

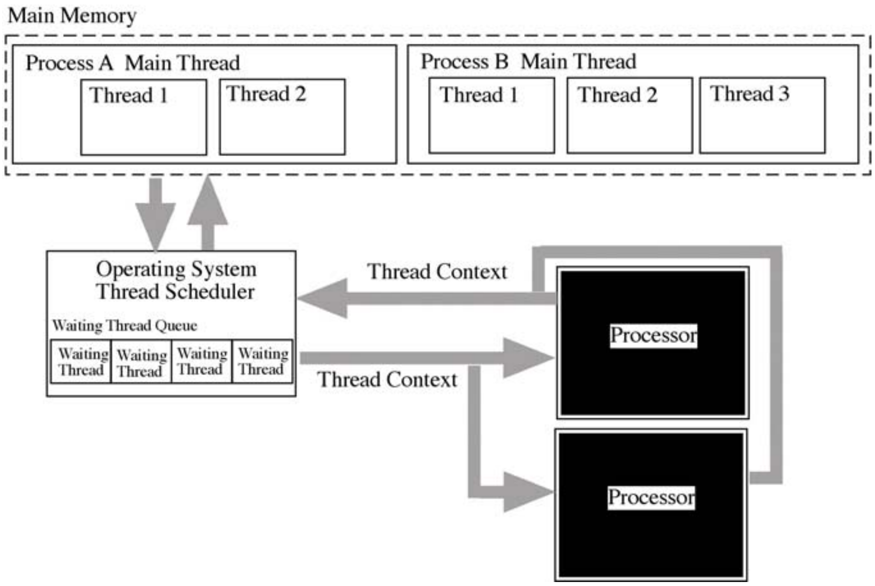


Figure 13-4: Processes and their Threads Executing in a Multiprocessor Environment

Quick Review

A process consists of one or more threads of execution, referred to simply as threads. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A thread is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a thread queue until they are loaded into the processor. Each thread has a data structure known as a thread context. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a time-slicing scheme. Each thread gets a little bit of time to execute before being preempted by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

Creating Managed Threads With The Thread Class

In this section I will show you how to use the Thread class to create and manage the execution of threads in your programs. You'll find the Thread class, along with a whole lot of other useful stuff, in the System.Threading namespace. The Thread class allows you to create what are referred to as *managed threads*. They are called managed threads because you can directly manipulate each thread you create. You gain a lot of flexibility and power when you manage your own threads. However, with power and flexibility comes the responsibility of ensuring your threads behave well and properly handle exceptional conditions that may arise during their execution lifetime. This aspect of thread management gained increased importance in .NET 2.0 because, in most cases, unhandled exceptions lead to application termination.

The material in this section lays the foundation for the rest of the chapter. Once you understand the issues involved with creating and managing your own threads, you'll better understand why, in most cases, it's a good idea to let the runtime environment manage threads for you. However, before getting started, let's see just how little relaxing one does while on single-threaded vacation.

Single-Threaded Vacation Example

How might the single-threaded vacation analogy be implemented in source code? Example 13.1 offers one possible solution.

13.1 SingleThreadedVacation.cs

```
1    using System;
2
3    public class SingleThreadedVacation {
4
5      private bool hungry;
6      private bool thirsty;
7
8      public SingleThreadedVacation(){
9        hungry = true;
10       thirsty = true;
11     }
12
13     public void FetchDrink(){
14       int steps_to_the_bar = 1000;
15       for(int i=0; i<steps_to_the_bar*2; i++){
16       if((i%100) == 0){
17          Console.WriteLine();
18          Console.Write("Fetching Drinks");
19       } else{
20          Console.Write(".");
21          }
22       }
23       Console.WriteLine();
24       thirsty = false;
25     }
26
27     public void FetchFood(){
28       int steps_to_the_grill = 1000;
29       for(int i=0; i<steps_to_the_grill*2; i++){
30       if((i%100)==0){
31          Console.WriteLine();
32          Console.Write("Fetching Food");
33       } else{
34          Console.Write(".");
35        }
36       }
37       Console.WriteLine();
38       hungry = false;
39     }
40
41     public static void Main(){
42       SingleThreadedVacation stv = new SingleThreadedVacation();
43       Console.WriteLine("Relaxing!");
44       while(stv.hungry && stv.thirsty){
45       stv.FetchDrink();
46       stv.FetchFood();
47       Console.WriteLine("Relaxing!");
48    }
49     }
50   }
```

Referring to example 13.1 — the SingleThreadedVacation class contains two fields: hungry and thirsty, of type bool, which are initially set to true. It has two methods: FetchDrink() and FetchFood(). When each method is called, the `for` loop contained in each kills some time by "walking" the number of steps to the bar or grill and back again. Each method prints to the console a status message every 100 steps it takes.

The Main() method starting on line 41 starts by printing a message to the console saying it's "Relaxing!". It then enters the `while` loop where calls are made to FetchDrink() and FetchFood(). Since the whole program executes in a single thread of execution (*i.e.,* the Main() method's thread,) the FetchDrink() method must run to conclusion before the call to FetchFood() can be made. The FetchFood() method must then execute and return before the message "Relaxing!" can again be printed to the screen. Figure 13-5 shows SingleThreadedVacation in action.

Multithreaded Vacation Example

Let's now see how much more relaxing you can do on a multithreaded vacation. Example 13.2 gives the code for the MultiThreadedVacation class.

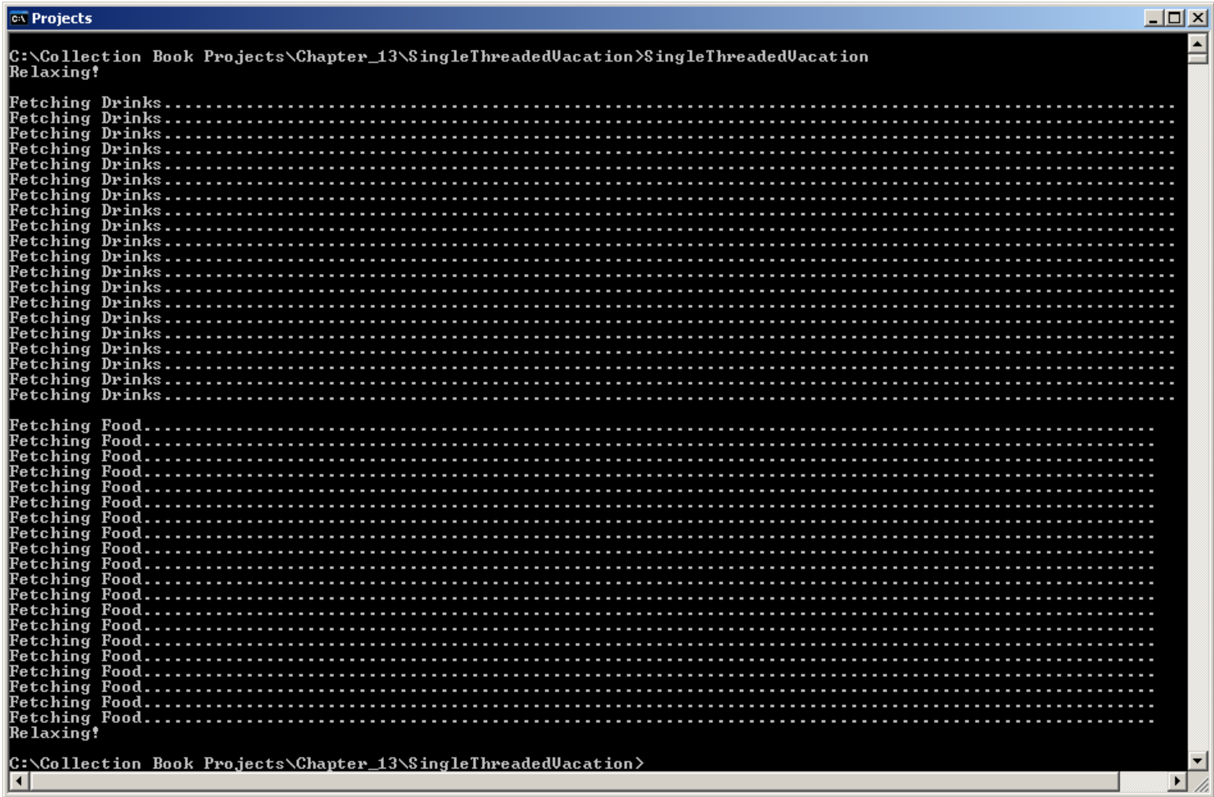


Figure 13-5: SingleThreadedVacation Program Output

13.2 MultiThreadedVacation.cs

```
using System;
using System.Threading;

public class MultiThreadedVacation {

  private bool hungry;
  private bool thirsty;

  public MultiThreadedVacation(){
    hungry = true;
    thirsty = true;
  }

  public void FetchDrink(){
    int steps_to_the_bar = 1000;
    for(int i=0; i<steps_to_the_bar*2; i++){
    if((i%100) == 0){
       Console.WriteLine();
       Console.Write("Fetching Drinks");
    } else{
       Console.Write(".");
     }
    }
    Console.WriteLine();
    thirsty = false;
  }

  public void FetchFood(){
    int steps_to_the_grill = 1000;
    for(int i=0; i<steps_to_the_grill*2; i++){
    if((i%100)==0){
       Console.WriteLine();
       Console.Write("Fetching Food");
    } else{
       Console.Write(".");
       }
    }
    Console.WriteLine();
```



```
39      hungry = false;
40    }
41
42    public static void Main(){
43      MultiThreadedVacation mtv = new MultiThreadedVacation();
44      Thread drinkFetcher = new Thread(mtv.FetchDrink);
45      Thread foodFetcher = new Thread(mtv.FetchFood);
46      Console.WriteLine("Relaxing!");
47
48      while(mtv.hungry && mtv.thirsty){
49         if(!drinkFetcher.IsAlive) drinkFetcher.Start();
50         if(!foodFetcher.IsAlive) foodFetcher.Start();
51         Console.Write("Relaxing!");
52      }
53    }
54  }
```

Referring to example 13.2 — this code is structurally very similar to the previous example. The only changes made were to the insides of the Main() method where two thread objects are created on lines 44 and 45 named drinkFetcher and foodFetcher respectively. Note that to create a thread in this fashion, you supply to the Thread constructor the name of a method you want to execute in the separate thread. (Here the method signatures conform to the ThreadStart delegate signature.) The drinkFetcher thread executes the FetchDrink() method while the foodFetcher thread executes the FetchFood() method.

A check is made in the body of the `while` loop to see if each thread is alive, meaning "Has it been started?" If not, it is started by calling its Thread.Start() method. As soon as these threads are started, the Main() thread can go back to printing the message "Relaxing!" to the console. Figure 13-6 shows a partial listing of the MultiThreadedVacation program's output. As you'll note from looking at figure 13-6 there's a lot more relaxing going on!

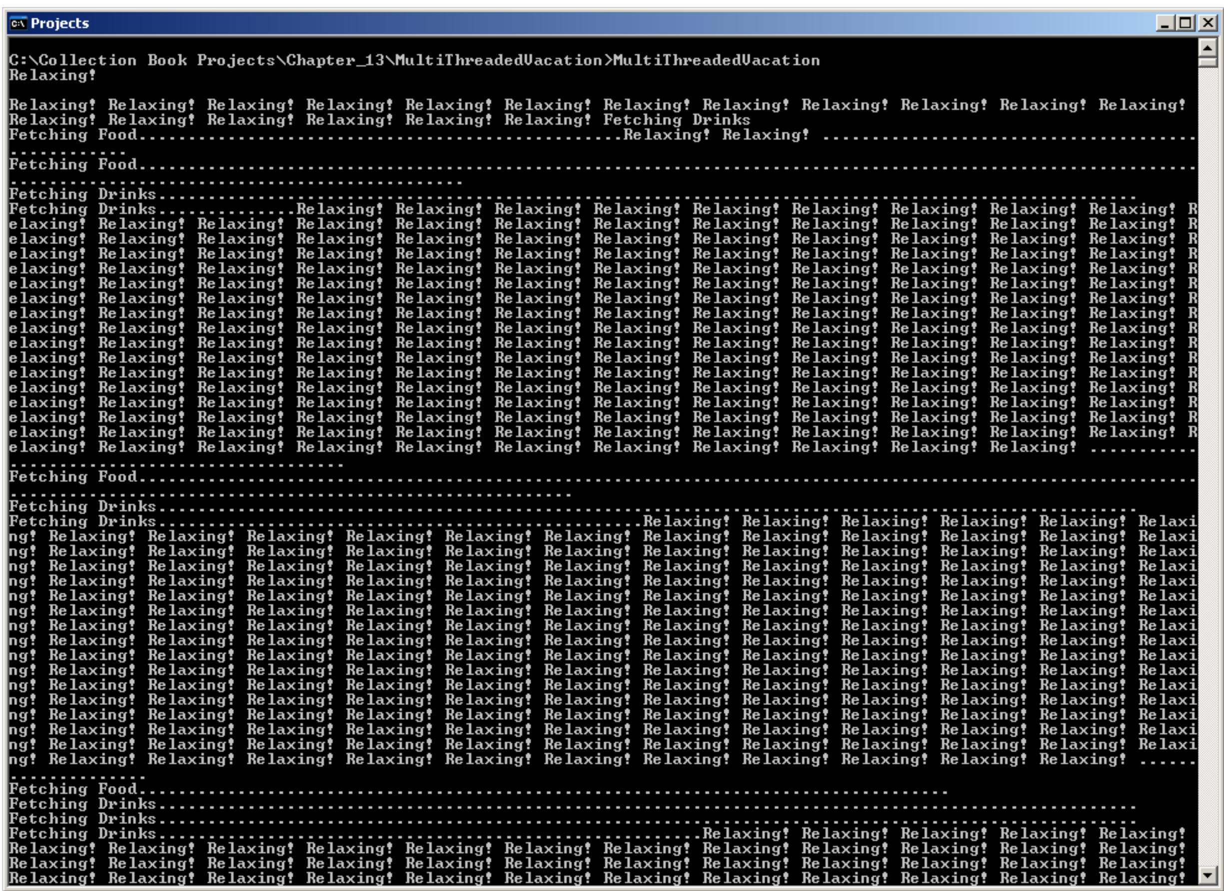


Figure 13-6: MultiThreadedVacation Program Output - Partial Listing

Thread States

A thread can assume several different states during its execution lifetime, as shown in figure 13-7.

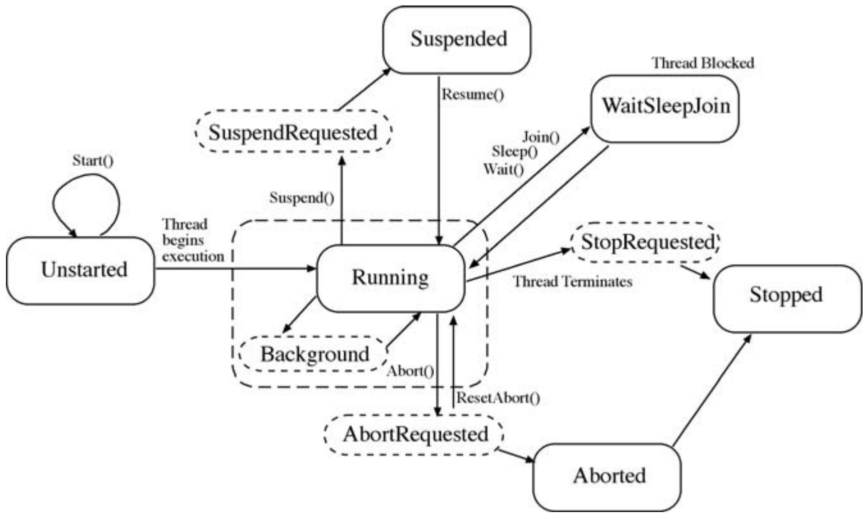


Figure 13-7: Thread States and Transition Initiators

Referring to figure 13-7 — important points to note include the following: A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

So where does that leave you with regards to managing your own threads? Well, you can start a thread with the Start() method and block its operation with the Monitor.Wait(), Thread.Sleep() and Thread.Join() methods. You can change a foreground thread into a background thread by setting its IsBackground property to true. As it turns out, this amount of control is really all you need to write well-behaved, multithreaded code. The following sections discuss and demonstrate the use of the more helpful Thread properties and methods.

Creating And Starting Managed Threads

To create a managed thread, pass in to the Thread constructor either a *ThreadStart* delegate or a *Parameter-izedThreadStart* delegate. The ParameterizedThreadStart delegate lets you pass an argument object when you call the thread's Start() method.

ThreadStart Delegate

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass a ThreadStart delegate into the Thread constructor: the *longhand* way and the *shorthand* way. The long-hand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

Example 13.3 demonstrates both the longhand and shorthand ways of creating threads.

13.3 ThreadStartDemo.cs

```
using System;
using System.Threading;

public class ThreadStartDemo {

  private const int COUNT = 200;

  public static void Run(){
    for(int i=0; i<COUNT; i++){
     Console.Write(Thread.CurrentThread.Name);
 }
  }

  public static void Main(){
  Thread thread1 = new Thread(new ThreadStart(Run)); // longhand way
  Thread thread2 = new Thread(Run); // shorthand way
  thread1.Name = "1";
  thread1.Start();
  thread2.Name = "2";
  thread2.Start();
  }
}
```

Referring to example 13.3 — two thread objects are created in the Main() method. The first, thread1, is created the longhand way by passing the name of the Run() method to the ThreadStart constructor. The second, thread2, is created the shorthand way by passing the name of the Run() method directly to the Thread constructor. Each thread's Name property is set before calling its Start() method. The name of the thread is printed to the console in the body of the Run() method. Note that in this example the Run() method is static, but it could just as well have been an instance method. Figure 13-8 shows the results of running this program.

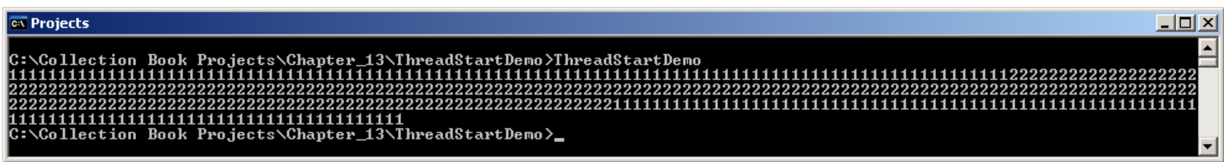


Figure 13-8: Results of Running Example 13.3

ParameterizedThreadStart Delegate: Passing Arguments To Threads

If you need to pass in an argument when you start a thread, your thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Example 13.4 shows the ParameterizedThreadStart delegate in action.

13.4 ParameterizedThreadStartDemo.cs

```
using System;
using System.Threading;

public class ParameterizedThreadStartDemo {

  private const int COUNT = 200;

  public static void Run(object value){
    for(int i=0; i<COUNT; i++){
    Console.Write(value);
 }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
  }
}
```


Referring to example 13.4 — The static Run() method has been modified to conform to the ParameterizedThreadStart delegate method signature. In this case I am passing the parameter named “value” directly to the Console.Write() method, which will automatically call the Object.ToString() method. (**Note:** Here I’m only targeting the interface as specified by the Object class. If I expect some other type of object I must cast to the expected type.) Pass the argument to the thread when you call its Start() method, as is shown on lines 17 and 18. Figure 13-9 shows the results of running this program.

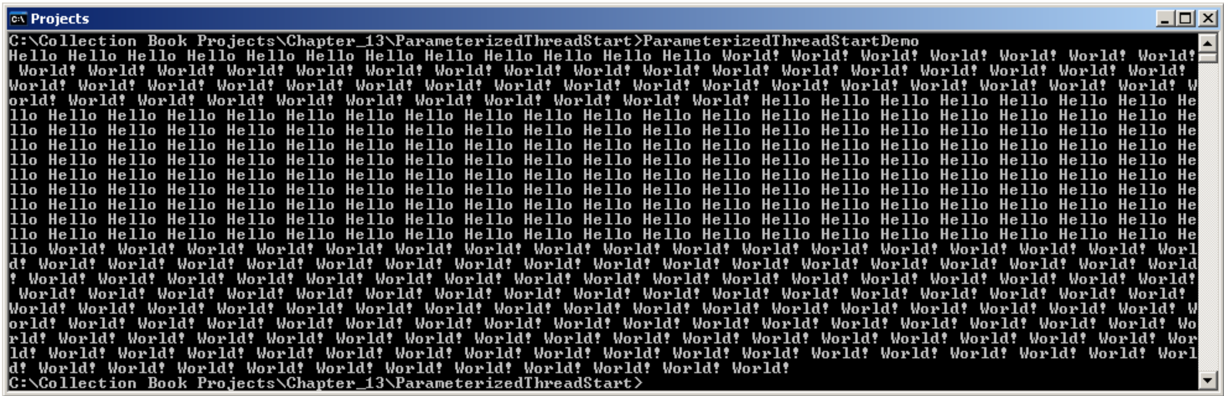


Figure 13-9: Results of Running Example 13.4

Blocking A Thread With Thread.Sleep()

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can’t finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Take a good look at figure 13-9 and you’ll see how thread1 prints the message “Hello” over and over until it’s swapped out with thread2, which then starts to print “World!”.

In many situations, you’ll want a thread to do something and then take a short break to let other threads have a go at the processor. Example 13.5 adds a call to Thread.Sleep() to the body of the Run() method.

13.5 ParameterizedThreadStart.cs (With call to Sleep())

```
using System;
using System.Threading;

public class ParameterizedThreadStartDemo {

  private const int COUNT = 200;

  public static void Run(object value){
    for(int i=0; i<COUNT; i++){
    Console.Write(value);
    Thread.Sleep(10);
 }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    hread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
  }
}
```

Referring to example 13.5 — the call to Thread.Sleep() is made after the value is written to the console. Pass an integer argument to the Sleep() method indicating the time in milliseconds you want the thread to block. You can also pass in a TimeSpan object. Figure 13-10 shows the results of running this program. Note how different the output appears and how much slower the application seems to run because of the increased thread swapping that occurs.

Figure 13-10: Results of Running Example 13.5

Blocking A Thread With Thread.Join()

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join(). I want to show you two examples to demonstrate the use of the Join() method. The first, example 13.6, builds on the previous example and adds a `for` loop in the Main() method that prints a message to the console. I've put the call to the thread2.Join() in the body of the `for` loop but it's commented out in this example.

13.6 JoinDemo.cs (Version 1)

```
1   using System;
2   using System.Threading;
3
4   public class JoinDemo {
5
6     private const int COUNT = 100;
7
8     public static void Run(object value){
9       for(int i=0; i<COUNT; i++){
10      Console.Write(value);
11      Thread.Sleep(10);
12   }
13    }
14
15    public static void Main(){
16      Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
17      Thread thread2 = new Thread(Run); // shorthand way
18      thread1.Start("Hello ");
19      thread2.Start("World! ");
20      for(int i = 0; i< 10; i++){
21         Console.Write("\n------- Main Thread Message --------");
22         //if(i==1) thread2.Join();
23      }
24    }
25  }
```

Referring to example 13.6 — I've added a `for` loop to the end of the Main() method that loops ten times printing a message to the console. I've commented out line 22 for now so you can compare the output of this program with the output of the next example. Figure 13-11 shows the results of running this program. Referring to figure 13-11 — note how thread1 and thread2 each print a message before sleeping. When the Main thread gets its chance to execute, it runs to completion.

Example 13.7 gives the JoinDemo program with line 22 in action. Figure 13-12 shows the results of running the program. Note the difference in the output between figures 13-11 and 13-12. The `for` loop in the Main thread makes it through two loops before being told to block until thread2 completes execution. (*i.e.,* thread2.Join())

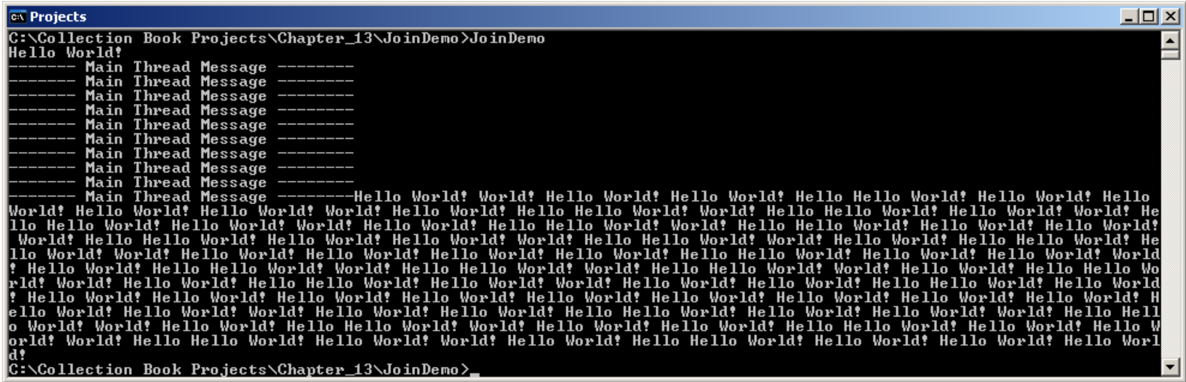


Figure 13-11: Results of Running Example 13.6

13.7 JoinDemo.cs (Version 2)

```
using System;
using System.Threading;

public class JoinDemo {

  private const int COUNT = 100;

  public static void Run(object value){
    for(int i=0; i<COUNT; i++){
    Console.Write(value);
    Thread.Sleep(10);
 }
  }

  public static void Main(){
    Thread thread1 = new Thread(new ParameterizedThreadStart(Run)); // longhand way
    Thread thread2 = new Thread(Run); // shorthand way
    thread1.Start("Hello ");
    thread2.Start("World! ");
    for(int i = 0; i< 10; i++){
       Console.Write("\n------- Main Thread Message --------");
       if(i==1) thread2.Join(); // the Main thread will block on thread2 after second loop
    }
  }
}
```

Figure 13-12: Results of Running Example 13.7

Foreground vs. Background Threads

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Managed threads are created as foreground threads. Example 13.8 gives an example of a foreground thread.

13.8 ForegroundThreadDemo.cs

```
using System;
using System.Threading;

public class ForegroundThreadDemo {

   public static void Run(){
    bool keepgoing = true;
    while(keepgoing){
      Console.Write("Please enter a letter or 'Q' to exit: ");
      String s = Console.ReadLine();
      switch(s[ 0] ){
         case 'Q': keepgoing = false;
                   break;
         default: break;
      }
    }
 }

  public static void Main(){
    Thread thread1 = new Thread(Run);
    thread1.Start();
  }
}
```

Referring to example 13.8 — the Main() method exits right after calling thread1.Start(). The Run() method loops continuously reading input from the console until the user enters the letter ‘Q’. Since thread1 is a foreground thread, it keeps the .NET runtime running as long as it’s executing. Figure 13-13 shows the results of running this program.

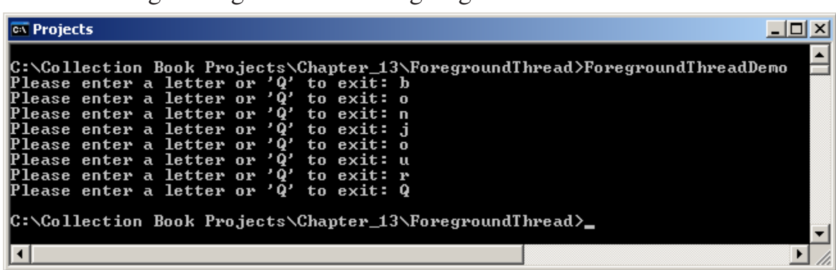


Figure 13-13: Results of Running Example 13.8

To change a foreground thread to a background thread, set the thread’s IsBackground property to true. Example 13.9 provides a slight modification to the previous example and makes thread1 a background thread.

13.9 BackgroundThreadDemo.cs

```
using System;
using System.Threading;

public class BackgroundThreadDemo {

   public static void Run(){
      bool keepgoing = true;
      while(keepgoing){
      Console.Write("Please enter a letter or 'Q' to exit: ");
      String s = Console.ReadLine();
      switch(s[ 0] ){
         case 'Q': keepgoing = false;
                   break;
         default: break;
      }
    }
 }

  public static void Main(){
    Thread thread1 = new Thread(Run);
    thread1.IsBackground = true;
    thread1.Start();
  }
}
```

Referring to example 13.9 — on line 21, thread1's IsBackground property is set to true. Its Start() method is called on the next line and the Main() method exits. Thus, thread1 is stopped along with the .NET runtime execution environment. Figure 13-14 shows the very brief results of running this program.

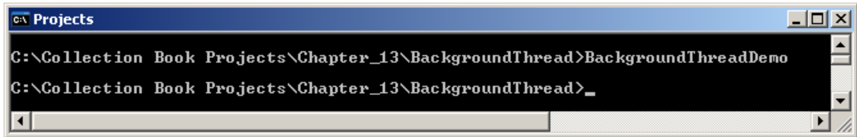


Figure 13-14: Results of Running Example 13.9

Quick Review

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() only notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in or, more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are now obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system *preempts* the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force your thread to block and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example, if you want the Main thread to block until thread2 completes execution, then in the Main thread you would call thread2.Join().

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread will keep the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Creating Threads With The BackgroundWorker Class

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads.

The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. Example 13.10 shows the BackgroundWorker class in action. This program displays a small window with three buttons and three labels. When you click one of the buttons it fires the background worker to do that particular task. The tasks, in this case, are to simply print a short message to the console and update the color of the label when the task starts running and when it completes.

13.10 BackgroundWorkerDemo.cs

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using System.ComponentModel;

public class BackgroundWorkerDemo : Form {

   private Button button1;
   private Button button2;
   private Button button3;
   private Label label1;
   private Label label2;
   private Label label3;
   private BackgroundWorker bw1;
   private BackgroundWorker bw2;
   private BackgroundWorker bw3;

   public BackgroundWorkerDemo(){
     InitializeComponents();
   }

   private void InitializeComponents(){
    button1 = new Button();
    button2 = new Button();
    button3 = new Button();
    label1 = new Label();
    label2 = new Label();
    label3 = new Label();
    bw1 = new BackgroundWorker();
    bw2 = new BackgroundWorker();
    bw3 = new BackgroundWorker();

    button1.Text = "Do Something";
    button1.AutoSize = true;
    button1.Click += ButtonOne_Click;
    label1.BackColor = Color.Green;
    bw1.DoWork += DoWorkOne;
    bw1.RunWorkerCompleted += ResetLabelOne;

    button2.Text = "Do Something Else";
    button2.AutoSize = true;
    button2.Click += ButtonTwo_Click;
    label2.BackColor = Color.Green;
    bw2.DoWork += DoWorkTwo;
    bw2.RunWorkerCompleted += ResetLabelTwo;

    button3.Text = "Do Something Different";
    button3.AutoSize = true;
    button3.Click += ButtonThree_Click;
    label3.BackColor = Color.Green;
    bw3.DoWork += DoWorkThree;
    bw3.RunWorkerCompleted += ResetLabelThree;
```



```

    TableLayoutPanel tlp1 = new TableLayoutPanel();
    tlp1.RowCount = 2;
    tlp1.ColumnCount = 3;
    tlp1.SuspendLayout();
    this.SuspendLayout();
    tlp1.AutoSize = true;
    tlp1.Dock = DockStyle.Left;
    tlp1.Controls.Add(button1);
    tlp1.Controls.Add(button2);
    tlp1.Controls.Add(button3);
    tlp1.Controls.Add(label1);
    tlp1.Controls.Add(label2);
    tlp1.Controls.Add(label3);
    this.Controls.Add(tlp1);
    this.AutoSize = true;
    this.AutoSizeMode = AutoSizeMode.GrowOnly;
    this.Height = tlp1.Height;
    tlp1.ResumeLayout();
    this.ResumeLayout();
  }

  private void ButtonOne_Click(Object sender, EventArgs e){
     if(!bw1.IsBusy){
       bw1.RunWorkerAsync(((Button)sender).Text);
     }
  }

  private void ButtonTwo_Click(Object sender, EventArgs e){
     if(!bw2.IsBusy){
       bw2.RunWorkerAsync(((Button)sender).Text);
     }
  }

  private void ButtonThree_Click(Object sender, EventArgs e){
     if(!bw3.IsBusy){
       bw3.RunWorkerAsync(((Button)sender).Text);
     }
  }

  private void DoWorkOne(Object sender, DoWorkEventArgs e){
     label1.BackColor = Color.Black;
     for(int i=0; i<30000; i++){
       Console.Write(e.Argument + " ");
     }
  }

   private void DoWorkTwo(Object sender, DoWorkEventArgs e){
     label2.BackColor = Color.Black;
     for(int i=0; i<30000; i++){
       Console.Write(e.Argument + " ");
     }
  }

   private void DoWorkThree(Object sender, DoWorkEventArgs e){
     label3.BackColor = Color.Black;
     for(int i=0; i<30000; i++){
       Console.Write(e.Argument + " ");
     }
  }

  private void ResetLabelOne(Object sender, RunWorkerCompletedEventArgs e){
     label1.BackColor = Color.Green;
  }

  private void ResetLabelTwo(Object sender, RunWorkerCompletedEventArgs e){
     label2.BackColor = Color.Green;
  }

  private void ResetLabelThree(Object sender, RunWorkerCompletedEventArgs e){
     label3.BackColor = Color.Green;
  }


 [STAThread]
 public static void Main(){
   Application.Run(new BackgroundWorkerDemo());
 }// end Main

} // end class definition
```


Referring to example 13.10 — in this code I create three buttons, three labels, and three BackgroundWorker objects named bw1, bw2, and bw3 respectively. To each background worker's *DoWork* event I assign a method that conforms to the *DoWorkEventHandler* delegate. These methods are named DoWorkOne(), DoWorkTwo(), and DoWorkThree(). To each background worker's *RunWorkerCompleted* event I assign a method that conforms to the *RunWorkerCompletedEventHandler* delegate. I named these methods ResetLabelOne(), ResetLabelTwo(), and ResetLabelThree().

To each button's Click event I assign methods that conform to the EventHandler delegate. I've named these methods ButtonOne_Click(), ButtonTwo_Click(), and ButtonThree_Click(). A click on each button calls its assigned event handler method. The event handler method kicks off a background worker thread by calling its RunWorkAsync() method. In this case, I'm passing in to the call to the RunWorkAsync() method the text of the clicked button.

A call to a background worker's RunWorkAsync() method fires its DoWork event. Any DoWorkEventHandlers assigned to the background worker's DoWork event are then called. Before actually making the call to RunWorkerAsync(), I check to see if the background worker is busy by polling its IsBusy property. If the background worker is currently running an asynchronous operation, the IsBusy property returns true.

When the background worker thread completes, its RunWorkerCompleted event fires resulting in a call to any assigned RunWorkerCompletedEventHandler methods. Figure 13-15 shows the results of running this program.

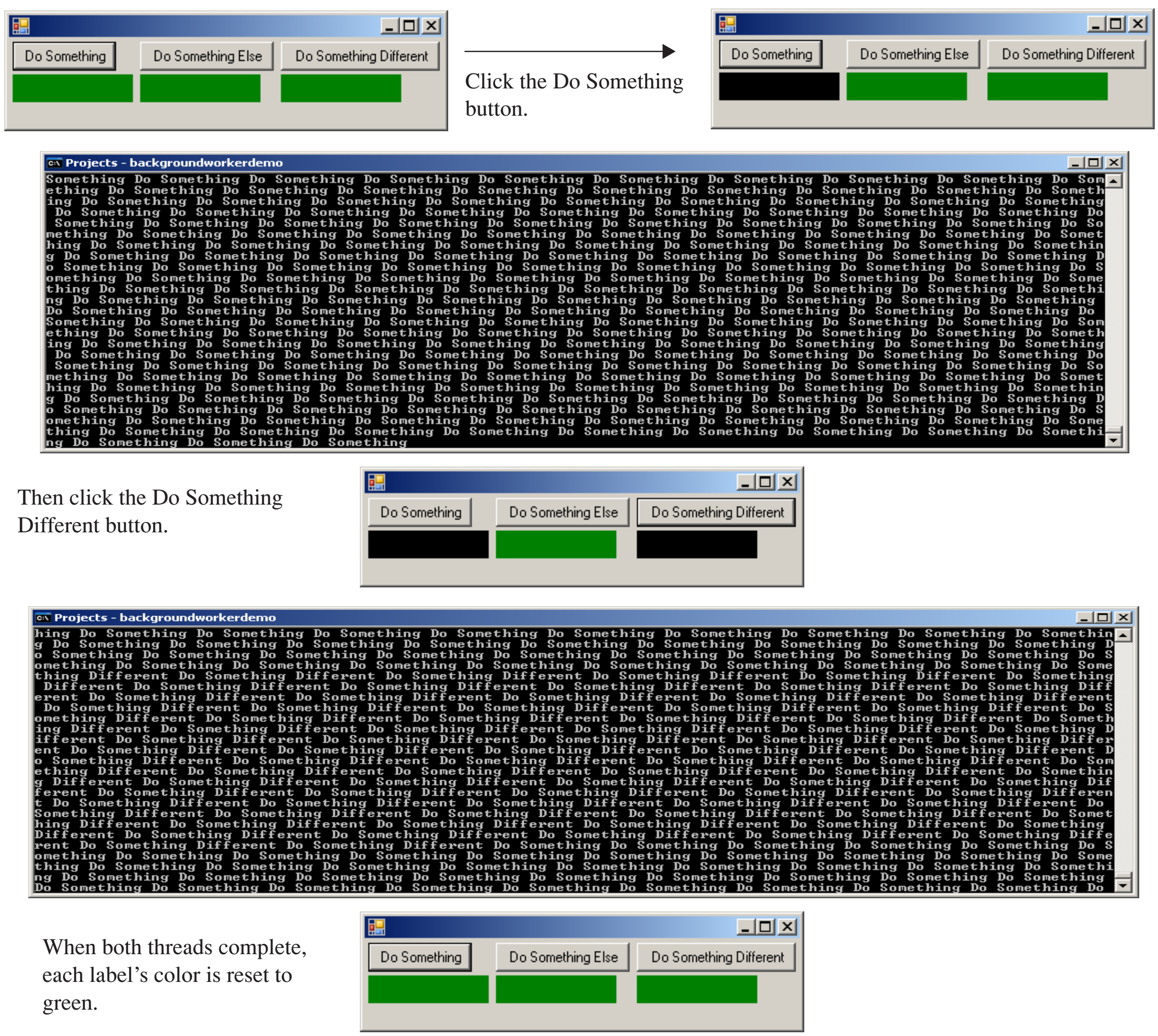


Figure 13-15: One Particular Result of Running Example 13.10

Quick Review

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.*BackgroundWorker* class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

Thread Pools

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class.

Beginning with .NET 2.0 Service Pack 1, each application's thread pool contains, by default, 250 worker threads per processor and 500 I/O completion port threads per processor. In this section, I will only show you how to use thread pool worker threads.

Important things to know about the application thread pool include the following:

- The ThreadPool class is static; its functionality is meant only to be used via its static methods.
- You can adjust the maximum number of threads in the pool via the ThreadPool.SetMaxThreads() method.
- To start a thread, pass the name of an execution method to the ThreadPool.QueueUserWorkItem() method.
- The thread pool contains a certain number of idle threads that are ready to execute. This number is adjusted via the ThreadPool.SetMinThreads() method. Too many idle threads extract a performance penalty because each idle thread requires stack space and other resources.
- The creation of new ThreadPool threads is throttled to one every 500 milliseconds. If you are spawning a large number of threads, you'll need to keep this throttling activity in mind.
- ThreadPool managed threads are background threads and will be terminated when your application exits.
- You have no control over a ThreadPool thread other than its initial creation.

Example 13.11 shows how easy it is to use ThreadPool threads. In this example, I spawn 45 separate threads with the help of the ThreadPool class. Following the creation of each thread, I print out the number of available threads.

13.11 ThreadPoolDemo.cs

```
using System;
using System.Threading;

public class ThreadPoolDemo {

  private const int COUNT = 20000;

  public static void Run(object stateInfo){
  for(int i=0; i<COUNT; i++){
  Console.Write(stateInfo + " ");
  Thread.Sleep(100);
 }
  }

  public static void Main(){
    int workerThreads = 0;
    int completionPortThreads = 0;
    ThreadPool.GetMinThreads(out workerThreads, out completionPortThreads);
    Console.WriteLine("Minimum number of worker threads in thread pool: {0} ", workerThreads);
    Console.WriteLine("Minimum number of completion port threads in thread pool: {0} ",
                         completionPortThreads);
    ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
    Console.WriteLine("Available worker threads in thread pool: {0} ", workerThreads);
    Console.WriteLine("Available completion port threads in thread pool: {0} ", completionPortThreads);

    for(int i = 0; i<45; i++){
      ThreadPool.QueueUserWorkItem(new WaitCallback(Run), i);
```



```
28        Thread.Sleep(1000); // sleep twice as long as it takes to start a threadpool thread
29        ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
30        Console.Write("\nAvailable worker threads in thread pool: {0} ", workerThreads);
31        Console.WriteLine("\nAvailable completion port threads in thread pool: {0}", completionPortThreads);
32      }
33    } // end Main() method
34  } // end class definition
```

Referring to example 13.11 — each new thread is created in the body of the `for` loop that begins on line 26. Note on line 27 that the ThreadPool.QueueUserWorkItem() method requires a WaitCallBack object. I have supplied the name of the thread execution method to the WaitCallBack constructor and pass the resulting object as an argument to the QueueUserWorkItem() method. On line 28, I put the Main() method thread to sleep for twice as long as it takes to create a new ThreadPool thread, and then print the number of available threads to the console.

In this example, I have modified the signature of the Run() method to conform to the WaitCallBack delegate. This allows me to pass arguments to the Run() method when I kick off each thread with the QueueUserWorkItem() method.

Figure 13-16 shows a partial result of running this program.

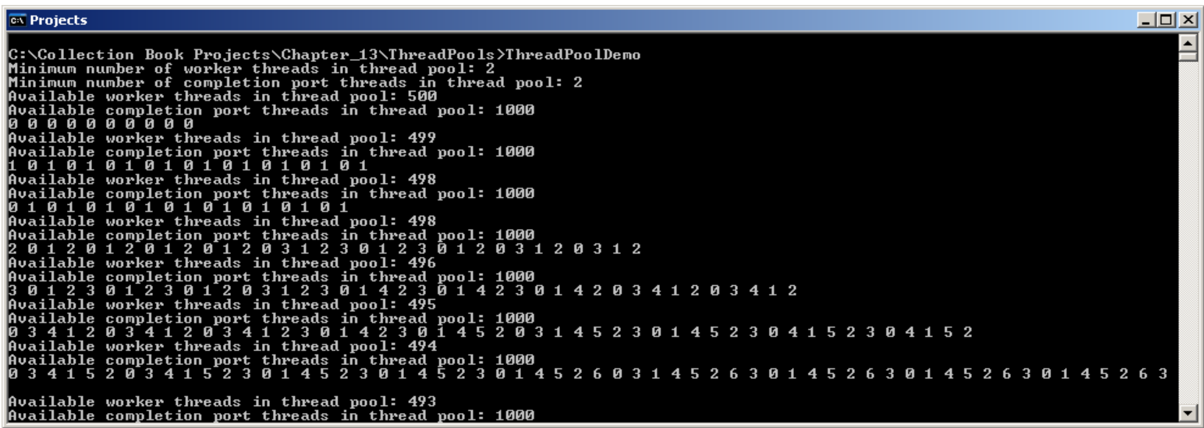


Figure 13-16: Partial Result of Running Example 13.11

Quick Review

The .NET runtime execution environment maintains and manages a pool of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of your thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Asynchronous Method Calls

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method asynchronously with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

To make an asynchronous method call follow these steps:

- Create a new delegate type that specifies the method signature of your thread execution method.
- Create your thread execution method making sure its method signature matches that of the delegate you created in the first step.
- Create an instance of the delegate, passing the name of the thread execution method to its constructor.

- Call the BeginInvoke() method on the delegate object, supplying any necessary thread execution method arguments and two additional arguments of type AsyncCallback and an Object respectively. I will discuss the purpose of the AsyncCallback and Object parameters shortly.
- The call to BeginInvoke() returns an IAsyncResult object that can be used to query the state of the asynchronous method call's execution progress. The IAsyncResult.AsyncState property is a reference to the last object supplied in the call to the BeginInvoke() method.
- Do any required work in the calling method while the asynchronous method call executes.
- Call the EndInvoke() method to properly wrap-up the asynchronous method call and fetch the results.

Example 13.12 shows the asynchronous call mechanism in action.

13.12 AsynchronousCallDemo.cs

```
1   using System;
2   using System.Threading;
3
4   public class AsynchronousCallDemo {
5
6     private const int COUNT = 100;
7     public delegate void RunDelegate(String message);
8
9     public static void Run(String message){
10      for(int i=0; i<COUNT; i++){
11        Console.Write(message + " ");
12        Thread.Sleep(100);
13      }
14    }
15
16    public static void Main(){
17      RunDelegate runDelegate1 = new RunDelegate(Run);
18      RunDelegate runDelegate2 = new RunDelegate(Run);
19      IAsyncResult result1 = runDelegate1.BeginInvoke("Hello", null, null);
20      IAsyncResult result2 = runDelegate2.BeginInvoke("World!", null, null);
21      while(!result1.IsCompleted && !result2.IsCompleted){
22        Console.Write(" - ");
23        Thread.Sleep(1000);
24      }
25      runDelegate1.EndInvoke(result1);
26      runDelegate2.EndInvoke(result2);
27      Console.WriteLine("\nMain thread exiting now...bye!");
28    } // end Main() method
29  } // end class definition
```

Referring to example 13.12 — on line 7, a new delegate type is declared named RunDelegate. The RunDelegate specifies a method that takes one String argument. The Run() method on line 9 conforms to the RunDelegate method signature specification. In the Main() method, I created two RunDelegate instances named runDelegate1 and runDelegate2. In the call to the RunDelegate constructor, I pass the name of the Run() method. I start the asynchronous methods by calling the BeginInvoke() method on each delegate instance passing in the required string argument and two null values representing the AsyncCallback and AsyncState objects, which are not being used in this case.

The `while` statement on line 21 loops until both IAsyncResult.IsCompleted properties are true. It prints the '-' character to the console and then sleeps for 1000 milliseconds to let the other two threads have a go at the processor.

On lines 25 and 26, the EndInvoke() method is called on each delegate instance, passing in the appropriate IAsyncResult reference. Figure 13-17 shows the results of running this program.

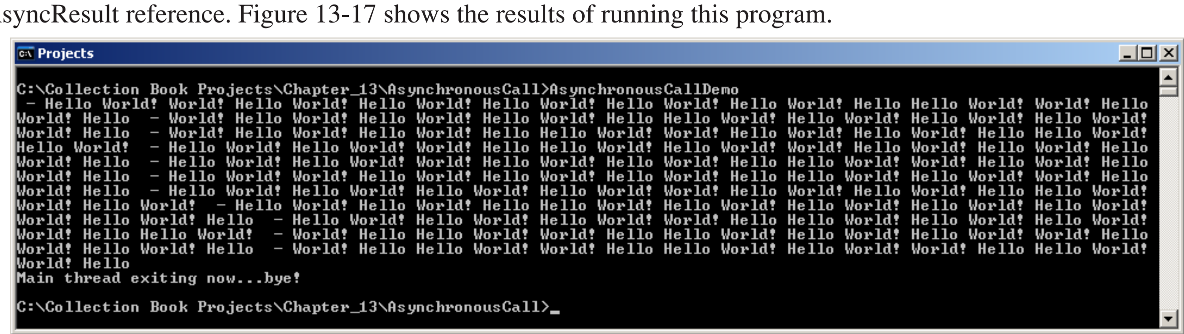


Figure 13-17: Results of Running Example 13.12

Obtaining Results From An Asynchronous Method Call

The are several ways to obtain results from an asynchronous method call. If the method returns a value, the call to the delegate's EndInvoke() method returns that value. If the method takes one or more `out` or `ref` parameters, these will be included in the EndInvoke() method's parameter list as well. (**Note:** Remember, a delegate's BeginInvoke() and EndInvoke() methods are automatically generated.) Example 13.13 demonstrates the use of the EndInvoke() method to retrieve an asynchronous method call's return value.

13.13 AsyncCallWithResultsDemo.cs

```
1   using System;
2   using System.Threading;
3
4   public class AsyncCallWithResultsDemo {
5
6     private const int COUNT = 100;
7     public delegate int SumDelegate(int a, int b);
8
9     public static int Sum(int a, int b){
10      return a + b;
11    }
12
13    public static void Main(){
14      SumDelegate sumDelegate1 = new SumDelegate(Sum);
15      SumDelegate sumDelegate2 = new SumDelegate(Sum);
16      IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, null, null);
17      IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, null, null);
18      while(!result1.IsCompleted && !result2.IsCompleted){
19        Thread.Sleep(100);
20      }
21      int sum1 = sumDelegate1.EndInvoke(result1);
22      int sum2 = sumDelegate2.EndInvoke(result2);
23      Console.WriteLine("The result of the first async method call is: {0}", sum1);
24      Console.WriteLine("The result of the second async method call is: {0}", sum2);
25      Console.WriteLine("\nMain thread exiting now...bye!");
26    } // end Main() method
27  } // end class definition
```

Referring to example 13.13 — I defined a delegate on line 7 named SumDelegate that takes two integer arguments and returns an integer value. The Sum() method on line 9 conforms to the SumDelegate signature. In the Main() method, two SumDelegate objects are created named sumDelegate1 and sumDelegate2. The BeginInvoke() method is called on each delegate. Note how the multiple arguments are passed to the asynchronous method call. On line 18, the `while` loop spins until both method calls complete, which in this case doesn't take too long because of the simplicity of the Sum() method. On lines 21 and 22, the results of each method call are obtained via the call to each delegate's EndInvoke() method and the values written to the console. Figure 13-18 shows the results of running this program.

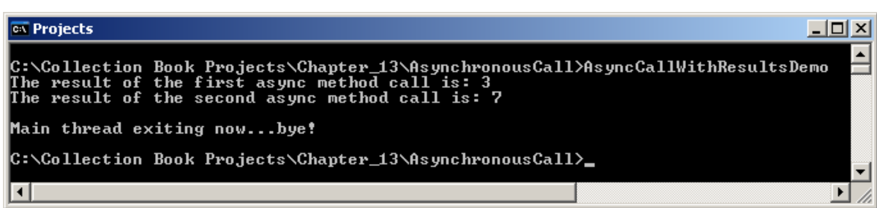


Figure 13-18: Results of Running Example 13.13

Providing A CallBack Method To BeginInvoke()

The BeginInvoke() method allows you to pass in a callback method that is automatically called when the asynchronous method completes execution. It also allows you to pass in an object argument to that callback method. Note that up until now I have been calling the BeginInvoke() method with the last two arguments set to null. *(i.e.,* sumDelegate1.BeginInvoke(1, 2, null, null)) To pass in a callback method, you'll need to write a method that conforms to the AsyncCallBack delegate method signature, which returns `void` and takes one argument of type IAsyncResult as the following code snippet shows:

```
void MethodName(IAsyncResult result)
```


The IAsyncResult interface specifies an AsyncState property of type Object, meaning it can contain any type of object. You can pass in whatever your heart desires! To use this object in the callback method, you'll need to access the IAsyncResult.AsyncState property and cast it to the expected type. Example 13.14 demonstrates the use of a callback method.

13.14 AsyncCallWithCallBackDemo.cs

```
1   using System;
2   using System.Threading;
3
4   public class AsyncCallWithCallBackDemo {
5
6     private const int COUNT = 100;
7     public delegate int SumDelegate(int a, int b);
8
9     public static int Sum(int a, int b){
10      return a + b;
11    }
12
13    public static void WrapUp(IAsyncResult result){
14      SumDelegate sumDelegate = (SumDelegate)result.AsyncState;
15      int sum = sumDelegate.EndInvoke(result);
16      Console.WriteLine("The result is: {0} ", sum);
17    }
18
19    public static void Main(){
20      SumDelegate sumDelegate1 = new SumDelegate(Sum);
21      SumDelegate sumDelegate2 = new SumDelegate(Sum);
22      IAsyncResult result1 = sumDelegate1.BeginInvoke(1, 2, new AsyncCallback(WrapUp), sumDelegate1);
23      IAsyncResult result2 = sumDelegate2.BeginInvoke(3, 4, new AsyncCallback(WrapUp), sumDelegate2);
24      while(!result1.IsCompleted && (!result2.IsCompleted)){
25        Console.WriteLine(" - ");
26        Thread.Sleep(10);
27      }
28      Console.WriteLine("\nMain thread exiting now...bye!");
29    } // end Main() method
30  } // end class definition
```

Referring to example 13.14 — I have added a method on line 13 named WrapUp() that conforms to the AsyncCallBack delegate method signature. In this example, I'm using the WrapUp() method to make the call to a SumDelegate's EndInvoke() method. To do this, I must pass in a reference to a SumDelegate, which I do as the last argument to each SumDelegate's BeginInvoke() method call shown on lines 22 and 23.

So, what's going on here? I'm executing two asynchronous method calls via two SumDelegate references. When each asynchronously executed method returns, the method supplied as the callback method is automatically called. It's a nice way to call and forget. However, since this is a simple console application, and the threads being created to execute the asynchronous method calls are thread pool background threads, the Main() method must hang on for a while and do some stuff, for if it exits right away, the background threads will be destroyed before they get a chance to execute. You generally don't have this problem when you're writing a GUI application. Figure 13-19 shows the results of running this program.

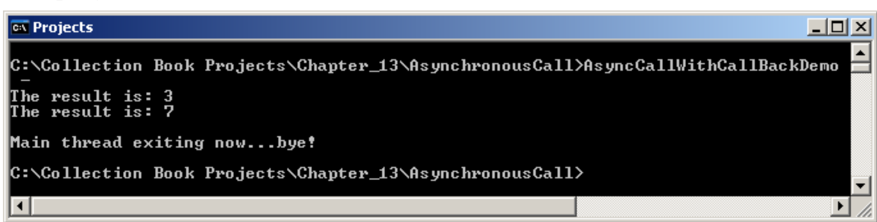


Figure 13-19: Results of Running Example 13.14

Quick Review

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a delegate. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

Summary

A *process* consists of one or more threads of execution, referred to simply as *threads*. A process always consists of at least one thread, the Main thread, which starts running when the process begins execution. A single-threaded process contains only one thread of execution. A multithreaded process contains more than one thread.

A *thread* is the smallest unit of code to which the operating system assigns processing time. A thread executes within the context of its containing or owning process and application domain.

Waiting threads sit in a *thread queue* until they are loaded into the processor. Each thread has a data structure known as a *thread context*. The thread context is a snapshot of the state of the processor and other execution details that must be preserved so that the thread can pick up execution where it left off when next loaded into the processor.

In a single-processor system the operating system allocates processor time with a *time-slicing* scheme. Each thread gets a little bit of time to execute before being *preempted* by the next waiting thread, at which point, if it's not finished with its business, it takes its place in the thread queue to wait another turn at the processor.

In a multiprocessor system, two threads can actually execute concurrently, but the operating system still uses time-slicing to manage their execution and keep the whole show running smoothly.

A thread can assume several different states during its execution lifetime. These states include: *Unstarted*, *Running*, *Background*, *SuspendRequested*, *Suspended*, *WaitSleepJoin*, *StopRequested*, *Stopped*, *AbortRequested*, and *Aborted*.

A call to a thread's Start() method does not immediately put the thread into the Running state. A call to Start() simply notifies the operating system that the thread can now be started. Also, a thread can be in multiple states simultaneously. For example, a Running thread can also be a Background thread, or a Suspended thread can also be in the AbortRequested state.

It's tricky at best to personally manage multiple threads by directly manipulating their states. In fact, Microsoft recommends you don't do it because it's hard to tell precisely what state a thread is actually in, or more importantly, at what point in the code the thread is at when you attempt to move it from one state to another. It's usually never a good idea to call Abort() on an executing thread, especially if you didn't start the thread. Another thing to consider is that the Suspend() and Resume() methods are obsolete.

To create a managed thread, pass to the Thread constructor either a ThreadStart delegate or a ParameterizedThreadStart delegate.

The ThreadStart delegate specifies a method signature that returns `void` and takes no arguments. There are two ways to pass the ThreadStart delegate to the Thread constructor: the *longhand* way and the *shorthand* way. The longhand way entails explicitly creating a new ThreadStart delegate object as the following code fragment suggests.

```
Thread thread1 = new Thread(new ThreadStart(Run)); // longhand
```

The shorthand method of creating a thread entails just passing the name of the method to the Thread constructor and letting it figure out if what you supplied conforms to the ThreadStart delegate as the following code fragment demonstrates:

```
Thread thread2 = new Thread(Run); // shorthand
```

If you need to pass in an argument when you start a thread, the thread's execution method must conform to the ParameterizedThreadStart delegate signature. The ParameterizedThreadStart delegate method signature is shown in following code fragment:

```
public void MethodName(object obj)
```

Like its ThreadStart delegate cousin, you can create threads the longhand or shorthand way. Pass the argument to the thread via its Start() method. Remember to cast the argument to the appropriate type in the body of the thread's execution method.

If all goes well, a thread, once started, charges forward and executes until it completes its assigned task. If it can't finish its assigned task in the allotted time slice, the operating system preempts the thread and swaps it out with another waiting thread. This swapping continues until the thread in question finishes its business or until something dreadful happens and it ends prematurely. Call the Thread.Sleep() method to force a thread to *block* and give other threads a chance to execute.

Another way to coordinate thread interaction is to explicitly block one thread until another thread completes execution. You can to this by calling the Thread.Join() method via the thread reference you want to yield to. For example,

if you want the Main thread to block until thread2 completes execution then in the Main thread you would call thread2.Join().

A thread can be either a *foreground* thread or a *background* thread. The difference being that a foreground thread keeps the .NET runtime alive so long as it is running. A background thread, however, will be shutdown by the .NET runtime when it shuts down.

Background threads are especially helpful when used with GUI applications as they allow time-intensive activities to proceed while minimizing the impact to the user interface experience. The System.ComponentModel.BackgroundWorker class makes it easy and convenient to create background threads that do heavy lifting behind the scenes while relieving you of the burden of explicitly managing those threads. The BackgroundWorker class provides this convenience and ease of use by allowing you to assign event handler methods to its various events. These events include *DoWork*, *ProgressChanged*, and *RunWorkerCompleted*. A call to a BackgroundWorker's RunWorkAsync() method fires its DoWork event.

The .NET runtime execution environment maintains and manages a *pool* of background threads for each application. You have access to these threads via the static methods of the ThreadPool class. By default, each application's thread pool contains 250 worker threads per processor and 500 I/O completion port threads per processor. Pass the name of the thread execution method to the WaitCallBack constructor; pass the WaitCallBack object to the ThreadPool.QueueUserWorkItem() method.

Multithreading is built into the very core of the .NET runtime execution environment. You can call any method *asynchronously* with the help of a *delegate*. You can do this via a delegate's BeginInvoke() and EndInvoke() methods. Don't go looking for these methods in the System.Delegate documentation; the .NET runtime environment creates them automatically when you declare and define a new delegate type. The thread that executes an asynchronous method comes from the application thread pool and is therefore a background thread.

References

Atul Gupta, *How Many Threads Have I Got?*, [http://infosysblogs.com/microsoft/2007/04/how_many_threads_have_i_got.html]

Steve Carr, et. al, *Race Conditions: A Case Study*

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Rick Miller, *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. 2008. ISBN: 1-932504-07-9

Notes

Chapter 14

Contax T

Waiting for the Orange Line

Collections And Threads

Learning Objectives

- *Understand the requirements for thread synchronization when manipulating collections*
- *Understand the difference between the ICollection and ICollection<T> interfaces*
- *Explain the purpose of the SyncRoot and IsSynchronized properties*
- *Explain how to create a synchronized collection and why it's not thread safe*
- *Understand how to synchronize access via the Monitor.Enter() and Monitor.Exit() methods*
- *State the relationship between the Monitor class and the C# lock keyword*
- *Employ the C# lock keyword to lock an object for thread synchronization*
- *State the names of the three synchronized collections in the System.Collections.Generic namespace*

Introduction

If you intend to use collection classes in a multithreaded environment you'll need to know how to ensure that only one thread has access to a collection at any time. This holds especially true if the items within a collection might be modified and enumerated by multiple threads. Fortunately, coordinating or synchronizing multiple thread access to a collection is easy to do; unfortunately, with the evolution of the .NET framework, several different thread synchronization strategies exist and are still supported in the framework, which makes it confusing for developers, both novice and experienced, as to which thread synchronization strategies work and which ones don't.

In this chapter I will show you how to synchronize multiple thread access to a collection. I will show you how to use the ICollection's SyncRoot and IsSynchronized properties as well as the Synchronized() method provided by some collections that is used to create Synchronized collection instances. I'll also explain why some collections implement the ICollection interface, which publishes the SyncRoot and IsSynchronized properties, while other collection's don't and how to program around this idiosyncrasy of the .NET collections framework. I will also explain why the Synchronized() method doesn't guarantee thread safety when enumerating through the elements of a collection.

Next I'll demonstrate the use of the Monitor.Enter() and Monitor.Exit() methods. I'll show you how to use the Monitor class in conjunction with a try/catch/finally block to ensure you exit the monitor. Following this I'll show you how to use the C# `lock` keyword to lock access to a collection using a separate lock object.

Some of the material I discuss in this chapter is deprecated in favor of more robust means of thread synchronization. I'm referring specifically to the reliance upon the SyncRoot and IsSynchronized properties of the ICollection interface and the use of synchronized collections created with the Synchronized() method found in some old-school, non-generic collection types. I present this material so that you better understand what you see when you dive into the .NET framework documentation and to increase your awareness of what has come before.

Also, I make no attempt to cover all aspects of thread synchronization. Specifically, I will omit coverage of WaitHandles, Mutexes, and the lightweight synchronization types introduced in .NET 4.0.

When you've finished this chapter you will have a clear understanding of how to apply a simple, effective thread synchronization strategy you can use to ensure thread-safe access to your collection objects. You'll also have a short list of simple rules to follow when implementing thread synchronization.

The Need For Thread Synchronization

If all you ever wanted to do was to read from a collection in a single-threaded environment then you could very well skip this chapter, and so could I, but that's not why you bought this book, so I'll keep typing.

Generally speaking, if your code is going to execute in a multi-threaded environment and multiple threads may execute *shared code segments* or access *shared resources or objects*, you'll want to control and coordinate access to these *critical code sections* by employing *thread synchronization mechanisms* provided by both the .NET framework and the C# language. However, not all thread synchronization mechanisms work as expected and in fact some are downright misleading. And, to make matters worse, the .NET framework has evolved and what was once provided for synchronization for the classes in the Collections namespace has been inconsistently carried forward and applied to the System.Collections.Generic classes. I'll talk more about this particular issue in another section titled: *SyncRoot, IsSynchroinzed, and Synchronized()*. Right now, I want to show you why thread synchronization is important, especially when multiple threads are trying to access and perhaps modify a collection's elements.

When might multiple threads need access to the same collection? The obvious scenario is when one thread is inserting objects into a collection and another thread is enumerating the collection at the same time. Example 14.1 offers a short program that demonstrates this scenario.

14.1 UnSynchronizedDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;

public class UnSynchronizedDemo {

   private List<int> _list = new List<int>();
```



```
8      private Random _random = new Random();
9      private const int ITEM_COUNT = 50;
10
11     public void InserterMethod(){
12       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
13       try {
14         for(int i=0; i<ITEM_COUNT; i++){
15           _list.Add(_random.Next(500));
16         }
17
18         Thread.Sleep(10);
19
20         for(int i=0; i<ITEM_COUNT; i++){
21           _list.Add(_random.Next(500));
22         }
23       } catch(Exception e){
24         Console.WriteLine(e);
25       }
26       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
27     }
28
29
30     public void ReaderMethod(){
31       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
32
33       try{
34         foreach(int i in _list){
35           Console.Write(i + " ");
36           Thread.Sleep(10);
37         }
38       } catch(Exception e){
39         Console.WriteLine(e);
40       }
41
42       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
43     }
44
45     public static void Main(){
46       UnSynchronizedDemo usd = new UnSynchronizedDemo();
47       Thread t1 = new Thread(usd.InserterMethod);
48       Thread t2 = new Thread(usd.ReaderMethod);
49       t1.Name = "Inserter Thread";
50       t2.Name = "Reader Thread";
51       t1.Start();
52       t2.Start();
53       t1.Join();
54       t2.Join();
55     }
56   }
```

Referring to example 14.1 — the UnSynchronzedDemo class declares and initializes a generic List<int> field named _list, a Random field named _random, and an integer constant named ITEM_COUNT. It defines two methhods: the first on line 11 named InserterMethod() and the second on line 30 named ReaderMethod(). The InserterMethod() steps through the _list with a `for` statement inserting random values between 0 and 500. It then calls the Thread.Sleep() method on line 18 to pause for a moment before again inserting values into the _list with a second `for` loop.

The ReaderMethod() uses the `foreach` statement to iterate over the _list elements. As you know by now the `foreach` statement accesses a collection's enumerator.

The Main() method on line 45 creates an instance of the UnSynchronizedDemo class named usd and then creates two separate threads named t1 and t2. Thread t1 runs the InserterMethod and thread t2 runs the ReaderMethod. On lines 49 and 50 I name each thread appropriately and then start each thread. The calls to t1.Join() and t2.Join() signal the Main thread to pause until threads t1 and t2 have finished executing before exiting.

What will happen in this program depends on timing and the amount of items being inserted into the collection by the Inserter thread t1. It may execute normally or it may throw an exception. If run enough times you'll get either result, but mostly you'll get an exception because the Inserter thread is trying to modify the _list during the enumeration performed by the Reader thread. Figure 14-1 shows the usual result of running this program.

Referring to figure 14-1 — as the console output shows, the Inserter thread starts execution first followed by the Reader thread, which managed to print two numbers to the console before the Inserter thread again started to insert numbers into the _list, which caused the exception. To prevent the exception you'll need to coordinate access to the collection by using thread synchronization so that only one thread has access to the collection at any time. The following section shows how to use the C# lock keyword to synchronize thread access to a collection.

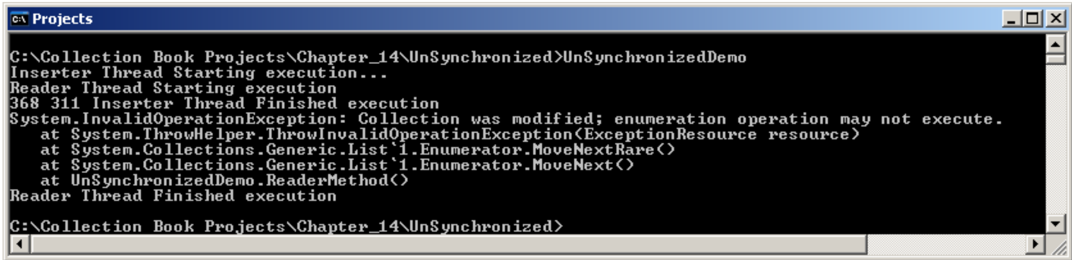


Figure 14-1: Results of Running Example 14.1

Quick Review

The need for thread synchronization arises when multiple threads of execution may access shared resources or shared code segments, which, if unsynchronized, would destabilize the code or leave the code in an invalid state. The .NET framework and the C# language provide various thread synchronization primitives and strategies that enable you to synchronize thread access to critical code segments.

Using The C# lock Keyword

The easiest way to implement thread synchronization is to use the C# lock keyword to obtain what is referred to as a "lock" on a particular object before entering a critical code section. Example 14.2 demonstrates the use of the lock keyword.

14.2 SynchronizedWithLockDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;

public class SynchronizedWithLockDemo {
   private List<int> _list = new List<int>();
   private Random _random = new Random();
   private const int ITEM_COUNT = 50;

   public void InserterMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
     Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
     lock(_list){
       Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
       for(int i=0; i<ITEM_COUNT; i++){
         _list.Add(_random.Next(500));
       }

       Thread.Sleep(10);

       for(int i=0; i<ITEM_COUNT; i++){
         _list.Add(_random.Next(500));
       }
     }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }

   public void ReaderMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
      Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
     lock(_list){
       Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
       foreach(int i in _list){
         Console.Write(i + " ");
         Thread.Sleep(10);
       }
    }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }

   public static void Main(){
```



```
      SynchronizedWithLockDemo swld = new SynchronizedWithLockDemo();
      Thread t1 = new Thread(swld.InserterMethod);
      Thread t2 = new Thread(swld.ReaderMethod);
      t1.Name = "Inserter Thread";
      t2.Name = "Reader Thread";
      t1.Start();
      t2.Start();
      t1.Join();
      t2.Join();
    }
  }
```

Referring to example 14.2 — this program is the same as example 14.1 except that in the InserterMethod() and the ReaderMethod() access to the _list collection is synchronized with the use of the `lock` keyword. I've also added several more diagnostic console output statements to help trace the program's execution.

Note how the `lock` keyword is used. The `lock` keyword takes a reference to an object as an argument. The critical section is denoted by the opening and closing braces. In this example I'm using the _list itself as the lock object, which is perfectly fine.

The important thing to note is that **all threads you wish to synchronize must lock the same object**. I put this last phrase in bold because it's important. It does no good to try to synchronize access using different lock objects, as you'll see later when I show you how thread synchronization works under the covers.

Figure 14-2 shows the results of running this program.

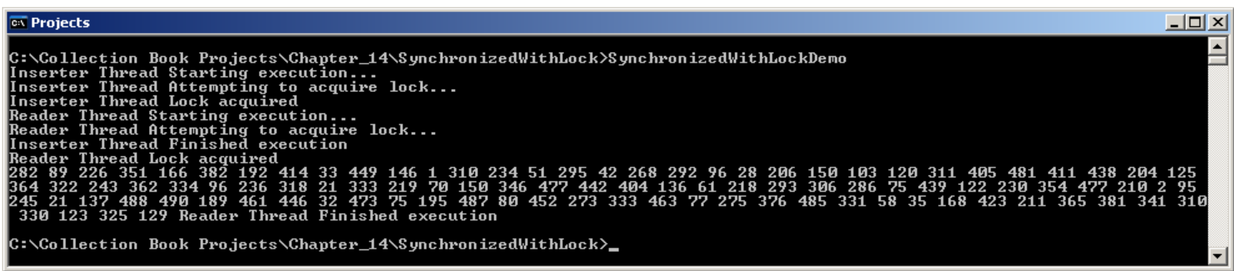


Figure 14-2: Results of Running Example 14.2

Referring to figure 14-2 — when the Inserter thread starts execution it immediately attempts to obtain the lock on the _list object. When the lock is acquired, the Inserter method enters the critical section. The Reader thread then starts execution and attempts to acquire the lock, but since the lock is held by the Inserter thread, it must wait until the Inserter thread completes and releases the lock on the _list object.

Note that in this example each thread runs to completion once it acquires the lock. So long a the Inserter thread runs first there will be items in the collection to enumerate. On the other hand, if the Reader thread manages to run first the _list would be empty. Again, this all depends on thread timing. Generally speaking, since I call t1.Start() first, the t1 thread is first to begin execution. Later I'll show you how to implement fine-grained thread control to handle the case where the Reader thread runs first and finds the _list empty. Before I do that I want to show you how thread synchronization works under the covers in the .NET runtime.

Quick Review

The C# `lock` keyword is the easiest way to protect critical code segments. Use the C# `lock` keyword to obtain a "lock" on an object. Place the code you want to protect within the body of the `lock` statement. **Recommendation: Lock on private field objects only.** Do not lock on the current instance (i.e. `this`). **Warning: Do not lock on value objects.** Value object are boxed into objects when used in a `lock` statement. Thus, multiple threads "locking" on the same value object will actually be acquiring locks on different objects.

Anatomy Of .NET Thread Synchronization

Figure 14-3 shows a diagram of how thread synchronization is implemented in the .NET runtime. I drew this diagram after studying the Microsoft Shared Source Common Language Infrastructure 2.0 (SSCLI 2.0) code which you can download from Microsoft.com. (See the References section.) The SSCLI virtual machine (VM) is implemented in C++. The four files of particular interest include: *Object.h*, *Object.cpp*, *SyncBlock.h*, and *SyncBlock.cpp*.

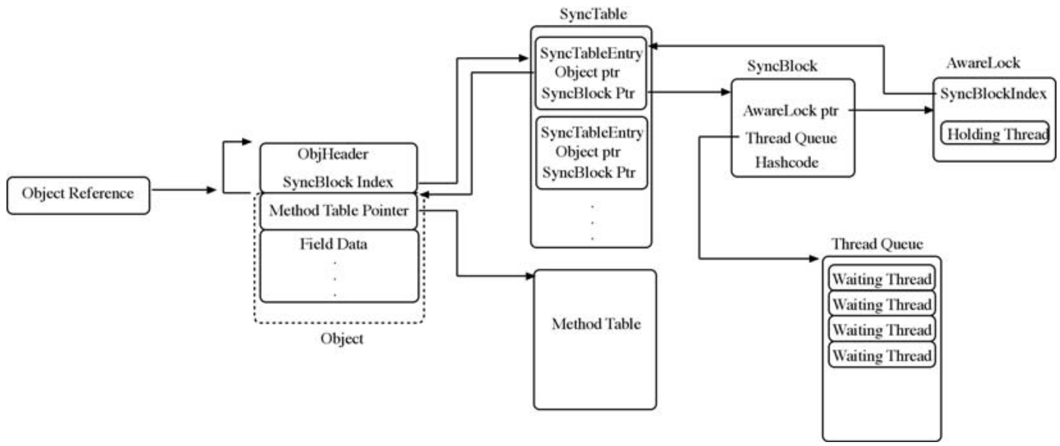


Figure 14-3: Thread Synchronization in the .NET Virtual Machine

Referring to figure 14-3 — the key players in thread synchronization include Object, ObjHeader, SyncTable, SyncTableEntry, SyncBlock, AwareLock, and ThreadQueue. Moving from left to right: an object reference points to an object instance within the virtual machine. This object instance is represented by the Object class as defined in the C++ virtual machine code. An object consists of a method table pointer and field data. At a negative offset from the beginning of the object is an object header (ObjHeader) which contains a data structure that, among other things, contains an index value to an entry into a SyncTable, which is an array of SyncTableEntry objects. For most objects in your program, the value of the SyncBlock index will be 0, meaning the object is not being used as a lock for a particular thread. When your code obtains a lock on a particular object, an unused SyncBlock is fetched from a SyncBlockCache (not shown in the diagram) and a SyncTableEntry is created in the SyncTable. The SyncTableEntry object has an object pointer that points back to the lock object, and a SyncBlock pointer that points to the SycnBlock. The SyncBlock object has a pointer to an AwareLock object and to a ThreadQueue which maintains a list of threads waiting to acquire the lock on the lock object. The bulk of the work is performed by the AwareLock class. Later, when you see how to use the Monitor.Enter() and Monitor.Exit() methods, it's the AwareLock object behind the scenes in the virtual machine that implements these methods as defined by the .NET System.Threading.Monitor class.

Old School – SyncRoot, IsSynchronized, and Synchronized()

The initial release of the .NET framework offered a confusing selection of properties and methods that gave developers a false sense of security with regards to thread synchronization. The ICollection interface provided the SyncRoot property which returns an object that can be used for thread synchronization. Most collections within the System.Collections namespace provide a Synchronized() method which is used to create a Synchronized collection instance. The IsSynchronized property simply returns true or false indicating whether or not a collection is synchronized.

The problem with creating and using a synchronized collection is that while access to certain parts of a collection's methods were synchronized, enumerating the collection's elements was not a thread safe operation. Studying the evolution of the .NET framework, which includes observing how developers learned to use .NET framework over the years since its release, leads me to conclude that it was developer confusion with regards to how to properly implement effective thread synchronization using the tools at hand, vs. any problems with the .NET thread synchronization tools per se.

Example 14.3 shows an example of a synchronized ArrayList created with the Array.Synchronized() method.

14.3 OldSchoolDemo.cs

```
1   using System;
2   using System.Threading;
3   using System.Collections;
4
5   public class OldSchoolDemo {
6      private ArrayList _list = new ArrayList();
7      private ArrayList _synchronizedList = null;
8      private const int ITEM_COUNT = 100;
9      private Random _random = new Random();
10
11     public OldSchoolDemo(){
12       _synchronizedList = ArrayList.Synchronized(_list);
13
14     }
15
16     public void PrintListStats(){
17       Console.WriteLine("The _list field IsSynchronized value: "
18                                         + _list.IsSynchronized);
19       Console.WriteLine("The _synchronizedList field IsSynchronized value: "
20                                         + _synchronizedList.IsSynchronized);
21     }
22
23     public void InserterMethod(){
24       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
25       for(int i=0; i<ITEM_COUNT; i++){
26         _synchronizedList.Add(_random.Next(500));
27       }
28
29       Thread.Sleep(10);
30
31       for(int i=0; i<ITEM_COUNT; i++){
32         _synchronizedList.Add(_random.Next(500));
33       }
34       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
35     }
36
37
38     private void ReaderMethod(){
39       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
40       try{
41         foreach(int i in _synchronizedList){
42           Console.Write(i + " ");
43         }
44       } catch(Exception e){
45         Console.WriteLine(e);
46       }
47       Console.WriteLine(Thread.CurrentThread.Name + " Finsihed execution...");
48     }
49
50    public static void Main(){
51      OldSchoolDemo osd = new OldSchoolDemo();
52      osd.PrintListStats();
53      Thread t1 = new Thread(osd.InserterMethod);
54      Thread t2 = new Thread(osd.ReaderMethod);
55      t1.Name = "Inserter thread";
56      t2.Name = "Reader thread";
57      t1.Start();
58      t2.Start();
59      t1.Join();
60      t2.Join();
61    }
62  }
```

Referring to example 14.3 — the OldSchoolDemo class declares and initializes an ArrayList named _list, an integer constant named ITEM_COUNT, and a Random object named _random. The initialization of _synchronizedList is performed in the body of the constructor. Note how the static method Array.Synchronized() is used to create the synchronized version of the array list. On line 16 the PrintListStats() method prints to the console the results obtained via calls to the IsSynchronized property on the _list and _synchronizedList.

The InserterMethod inserts random integers between the values 0 and 500 into the _list. It then sleeps for 10 milliseconds and then inserts more integers into the _list. The ReaderMethod uses the `foreach` method to print the list items to the console.

The Main() method creates two threads named t1 and t2. Thread t1 runs the InserterMethod and thread t2 runs the ReaderMethod(). Thread t1 is named Inserter and thread t2 is named Reader.

Figure 14-4 shows the results of running this program.

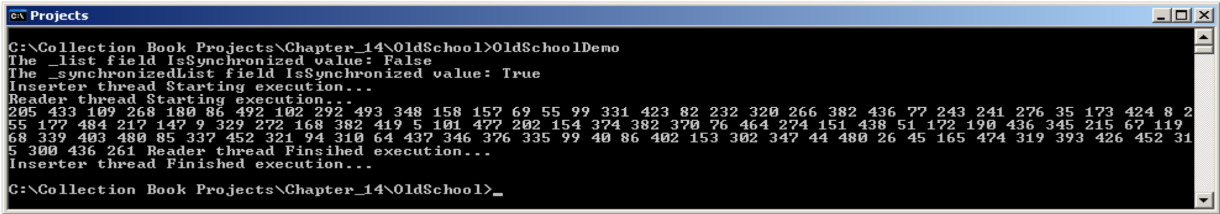


Figure 14-4: One Possible Result of Running Example 14.3

Referring to figure 14-4 — notice that the Inserter thread did not finish execution before the Reader thread started to run. It was by pure luck of timing that an exception was not thrown. Figure 14-5 shows the usual result of running this program repeatedly.

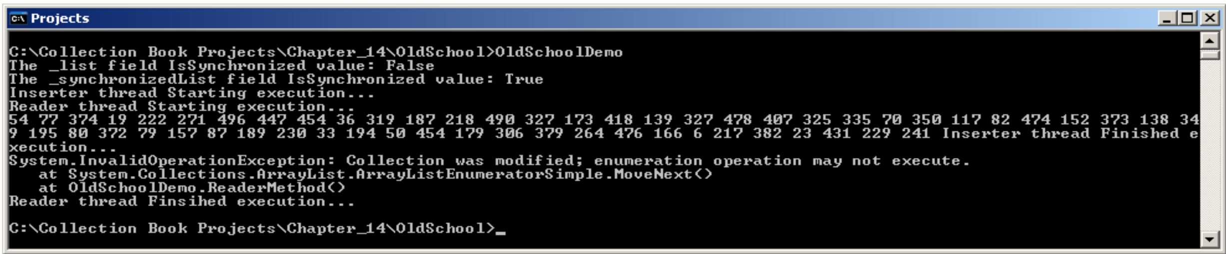


Figure 14-5: The Usual Result of Running Example 14.3

Even though the list is synchronized, you must still take steps to coordinate multithread access to it when enumerating its elements. Example 14.4 shows how the `lock` keyword could be used in conjunction with the _synchronizedList.SyncRoot property.

14.4 OldSchoolSyncRootDemo.cs

```
using System;
using System.Threading;
using System.Collections;

public class OldSchoolSyncRootDemo {
   private ArrayList _list = new ArrayList();
   private ArrayList _synchronizedList = null;
   private const int ITEM_COUNT = 50;
   private Random _random = new Random();

   public OldSchoolSyncRootDemo(){
     _synchronizedList = ArrayList.Synchronized(_list);

   }

   public void PrintListStats(){
     Console.WriteLine("The _list field IsSynchronized value: "
                                          + _list.IsSynchronized);
     Console.WriteLine("The _synchronizedList field IsSynchronized value: "
                                          + _synchronizedList.IsSynchronized);
   }

   public void InserterMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
      Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire the lock...");
     lock(_synchronizedList.SyncRoot){
       Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired...");
       for(int i=0; i<ITEM_COUNT; i++){
         _synchronizedList.Add(_random.Next(500));
       }

       Console.WriteLine(Thread.CurrentThread.Name + " Sleeping...");
       Thread.Sleep(10);

       for(int i=0; i<ITEM_COUNT; i++){
         _synchronizedList.Add(_random.Next(500));
       }
     }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
   }


   private void ReaderMethod(){
```



```
44       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
45       lock(_synchronizedList.SyncRoot){
46         try{
47           foreach(int i in _synchronizedList){
48             Console.Write(i + " ");
49             Console.Write(Thread.CurrentThread.Name + " Sleeping...");
50             Thread.Sleep(10);
51           }
52         } catch(Exception e){
53           Console.WriteLine(e);
54         }
55       }
56       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution...");
57     }
58
59    public static void Main(){
60      OldSchoolSyncRootDemo ossrd = new OldSchoolSyncRootDemo();
61      ossrd.PrintListStats();
62      Thread t1 = new Thread(ossrd.InserterMethod);
63      Thread t2 = new Thread(ossrd.ReaderMethod);
64      t1.Name = "Inserter thread";
65      t2.Name = "Reader thread";
66      t1.Start();
67      t2.Start();
68      t1.Join();
69      t2.Join();
70    }
71  }
```

Referring to example 14.4 — this code is similar to example 14.3 except now the `lock` keyword is being used to protect the critical section of the InserterMethod() and the ReaderMethod(). (Lines 26 and 45 respectively.) Note that in this case I'm locking on the _synchronizedList.SyncRoot property which is more than likely just a reference to the _synchronizedList object itself behind the scenes. Figure 14-6 shows one possible result of running this program.

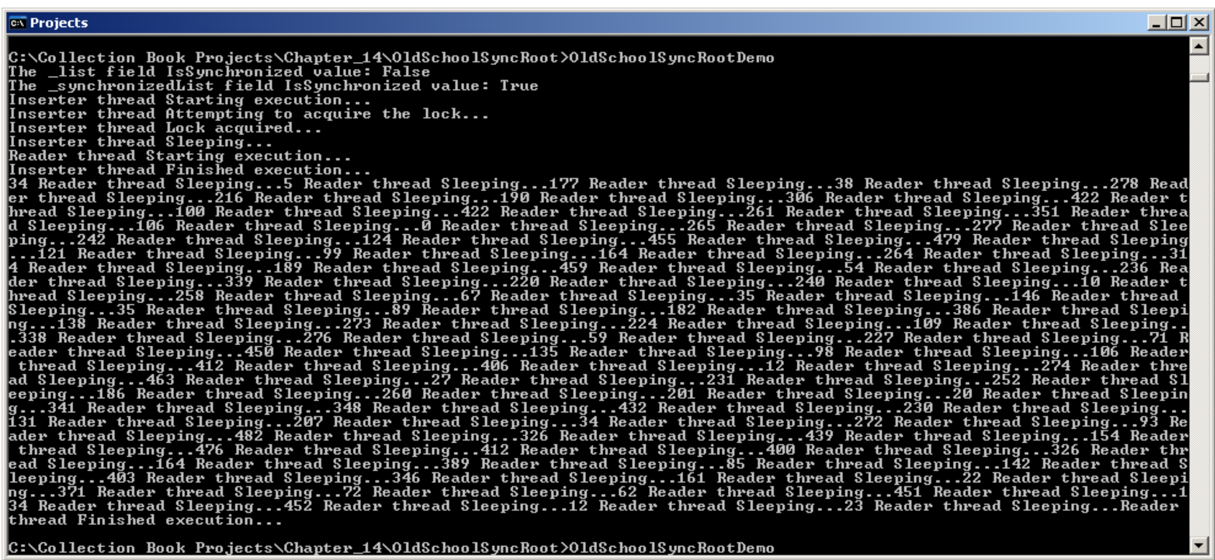


Figure 14-6: One Possible Result of Running Example 14.4

Again, depending on when thread t1 actually starts running, thread t2 may start to run before t1 acquires the lock and gets a chance to insert any items into the _synchronizedList. Figure 14-7 shows another possible result of running example 14.4.

Quick Review

Collection classes in the System.Collections namespace come equipped with the SyncRoot and IsSynchronized properties. These old-school collections also provided a static Synchronized() method which is used to transform an ordinary collection into a synchronized collection. And while individual collection methods may be synchronized, it was still not thread safe to enumerate over a collection. While you can still write good-quality thread-safe code using the SyncRoot property along with the `lock` keyword or the Monitor class, the use of these old-school properties,

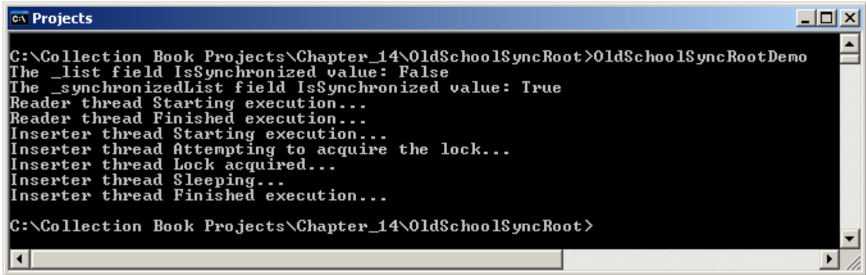


Figure 14-7: Another Possible Result from Running Example 14.4

along with the Synchronized() method is best avoided. Besides, unless you find yourself maintaining legacy C# code, you should be favoring the use of the generic collection classes.

Monitor.Enter() and Monitor.Exit()

The Monitor class can be used to synchronize thread access to critical code sections just like the C# `lock` keyword. In fact, the C# `lock` keyword is translated into Monitor.Enter() and Monitor.Exit() method calls by the compiler. Example 14.5 lists the decompiled intermediate language for the InserterMethod() of example 14.2.

14.5 Decompiled InserterMethod from Example 14.2

```
.method public hidebysig instance void  InserterMethod() cil managed
{
  // Code size       247 (0xf7)
  .maxstack  3
  .locals init (int32 V_0,
           bool V_1,
           class [mscorlib]System.Collections.Generic.List`1<int32> V_2,
           bool V_3)
  IL_0000:  nop
  IL_0001:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
  IL_0006:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
  IL_000b:  ldstr      " Starting execution..."
  IL_0010:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0015:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_001a:  nop
  IL_001b:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
  IL_0020:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
  IL_0025:  ldstr      " Attempting to acquire lock..."
  IL_002a:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_002f:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0034:  nop
  IL_0035:  ldc.i4.0
  IL_0036:  stloc.1
  .try
  {
    IL_0037:  nop
    IL_0038:  ldarg.0
    IL_0039:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32>
SynchronizedWithLockDemo::_list
    IL_003e:  dup
    IL_003f:  stloc.2
    IL_0040:  ldloca.s   V_1
    IL_0042:  call       void [mscorlib]System.Threading.Monitor::Enter(object,
                                                                     bool&)
    IL_0047:  nop
    IL_0048:  call       class [mscorlib]System.Threading.Thread
[mscorlib]System.Threading.Thread::get_CurrentThread()
    IL_004d:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
    IL_0052:  ldstr      " Lock acquired"
    IL_0057:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
    IL_005c:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0061:  nop
    IL_0062:  ldc.i4.0
```

```
      IL_0063:  stloc.0
      IL_0064:  br.s       IL_0088
      IL_0066:  nop
      IL_0067:  ldarg.0
      IL_0068:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32> SynchronizedWithLockDemo::_list
      IL_006d:  ldarg.0
      IL_006e:  ldfld      class [mscorlib]System.Random SynchronizedWithLockDemo::_random
      IL_0073:  ldc.i4     0x1f4
      IL_0078:  callvirt   instance int32 [mscorlib]System.Random::Next(int32)
      IL_007d:  callvirt   instance void class [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
      IL_0082:  nop
      IL_0083:  nop
      IL_0084:  ldloc.0
      IL_0085:  ldc.i4.1
      IL_0086:  add
      IL_0087:  stloc.0
      IL_0088:  ldloc.0
      IL_0089:  ldc.i4.s   50
      IL_008b:  clt
      IL_008d:  stloc.3
      IL_008e:  ldloc.3
      IL_008f:  brtrue.s   IL_0066
      IL_0091:  ldc.i4.s   10
      IL_0093:  call       void [mscorlib]System.Threading.Thread::Sleep(int32)
      IL_0098:  nop
      IL_0099:  ldc.i4.0
      IL_009a:  stloc.0
      IL_009b:  br.s       IL_00bf
      IL_009d:  nop
      IL_009e:  ldarg.0
      IL_009f:  ldfld      class [mscorlib]System.Collections.Generic.List`1<int32> SynchronizedWithLockDemo::_list
      IL_00a4:  ldarg.0
      IL_00a5:  ldfld      class [mscorlib]System.Random SynchronizedWithLockDemo::_random
      IL_00aa:  ldc.i4     0x1f4
      IL_00af:  callvirt   instance int32 [mscorlib]System.Random::Next(int32)
      IL_00b4:  callvirt   instance void class [mscorlib]System.Collections.Generic.List`1<int32>::Add(!0)
      IL_00b9:  nop
      IL_00ba:  nop
      IL_00bb:  ldloc.0
      IL_00bc:  ldc.i4.1
      IL_00bd:  add
      IL_00be:  stloc.0
      IL_00bf:  ldloc.0
      IL_00c0:  ldc.i4.s   50
      IL_00c2:  clt
      IL_00c4:  stloc.3
      IL_00c5:  ldloc.3
      IL_00c6:  brtrue.s   IL_009d
      IL_00c8:  nop
      IL_00c9:  leave.s    IL_00db
    }  // end .try
    finally
    {
      IL_00cb:  ldloc.1
      IL_00cc:  ldc.i4.0
      IL_00cd:  ceq
      IL_00cf:  stloc.3
      IL_00d0:  ldloc.3
      IL_00d1:  brtrue.s   IL_00da
      IL_00d3:  ldloc.2
      IL_00d4:  call       void [mscorlib]System.Threading.Monitor::Exit(object)
      IL_00d9:  nop
      IL_00da:  endfinally
    }  // end handler
    IL_00db:  nop
    IL_00dc:  call       class [mscorlib]System.Threading.Thread [mscorlib]System.Threading.Thread::get_CurrentThread()
    IL_00e1:  callvirt   instance string [mscorlib]System.Threading.Thread::get_Name()
    IL_00e6:  ldstr      " Finished execution"
    IL_00eb:  call       string [mscorlib]System.String::Concat(string,
                                                                  string)
    IL_00f0:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_00f5:  nop
    IL_00f6:  ret
  } // end of method SynchronizedWithLockDemo::InserterMethod
```


Referring to example 14.5 — the InserterMethod() in example 14.2 used the C# `lock` keyword to synchronize thread access to its critical section. Line 34 shows how the actual call is made to the Monitor.Enter() and later, on line 105 to Monitor.Exit().

Using Monitor.Enter() and Monitor.Exit()

While the C# `lock` keyword makes thread synchronization easy, the use of the Monitor class demands you pay more attention to what you're doing. You must be sure to call Monitor.Exit() for each call to Monitor.Enter(). The way to ensure this happens is to use the Monitor.Enter() and Monitor.Exit() methods in conjunction with a `try/catch/finally` block. Example 14.6 demonstrates the use of Monitor.Enter() and Monitor.Exit().

14.6 MonitorDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;

public class MonitorDemo {

   private List<int> _list = new List<int>();
   private Random _random = new Random();
   private const int ITEM_COUNT = 50;

   public void InserterMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
     Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
     Monitor.Enter(_list);
      Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
     try{
         for(int i=0; i<ITEM_COUNT; i++){
           _list.Add(_random.Next(500));
         }

          Console.WriteLine(Thread.CurrentThread.Name + " Sleeping...");
         Thread.Sleep(10);

         for(int i=0; i<ITEM_COUNT; i++){
           _list.Add(_random.Next(500));
         }
      }catch(Exception e){
         Console.WriteLine(e);
     }finally{
       Monitor.Exit(_list);
        Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
     }
      Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }

   public void ReaderMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");
     Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
     Monitor.Enter(_list);
     Console.WriteLine(Thread.CurrentThread.Name + " Lock acquired");
     try{
       foreach(int i in _list){
         Console.Write(i + " ");
          Console.Write(Thread.CurrentThread.Name + " Sleeping...");
         Thread.Sleep(10);
       }
     }catch(Exception e){
       Console.WriteLine(e);
     }finally{
       Monitor.Exit(_list);
        Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
     }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }


   public static void Main(){
     MonitorDemo md = new MonitorDemo();
     Thread t1 = new Thread(md.InserterMethod);
     Thread t2 = new Thread(md.ReaderMethod);
     t1.Name = "Inserter Thread";
     t2.Name = "Reader Thread";
     t1.Start();
     t2.Start();
```



```
65        t1.Join();
66        t2.Join();
67     }
68  }
```

Referring to example 14.6 — this program is similar to example 14.2 only the critical section in the InserterMethod() and ReaderMethod() is protected with the help of Monitor.Enter() and Monitor.Exit(). Note that a reference to the lock object is passed to both the Monitor.Enter() and Monitor.Exit() methods. (e.g., Monitor.Enter(_list) and Monitor.Exit(_list))

Let's take a closer look at the use of Monitor.Enter() and Monitor.Exit() in the body of the InserterMethod(). The call to Monitor.Enter(_list) is made on line 14. **The Monitor.Enter() method blocks until a lock is obtained.** This effectively stops execution of the current thread until the thread that owns the lock on _list, which in this example would be the ReaderMethod(), releases its lock on _list. Note too that the call to Monitor.Enter() marks the beginning of the critical section. Figure 14-8 shows the results of running this program.

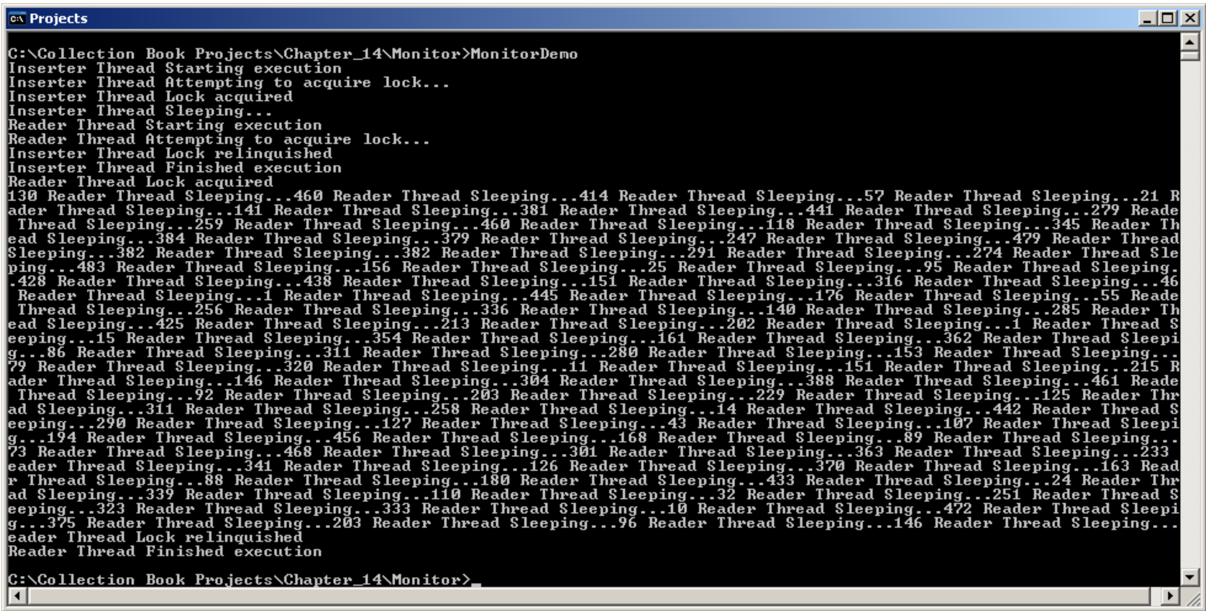


Figure 14-8: Results of Running Example 14.6

Using Overloaded Monitor.Enter() Method

The single-argument version of the Monitor.Enter() method is obsolete as of .NET 4.0 and it's recommended that going forward you use the overloaded version of the method which takes two arguments: a reference to a lock object and a boolean ref variable that is set to true if the lock is acquired. The use of the new overloaded Monitor.Enter() method comes with a new recommended usage structure as well. Example 14.7 demonstrates the use of the overloaded Monitor.Enter() method. This example also demonstrates the use of the Monitor.Wait() and Monitor.Pulse() methods.

14.7 MonitorLockTakenDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;

public class MonitorLockTakenDemo {

   private List<int>  _list = new List<int>();
   private Random _random = new Random();
   private const int ITEM_COUNT = 50;


   public void InserterMethod(){
    Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
     bool lockTaken = false;
     try{
       Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
```



```
17        Monitor.Enter(_list, ref lockTaken);
18        if(lockTaken){
19          Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
20          for(int i=0; i<ITEM_COUNT; i++){
21            _list.Add(_random.Next(500));
22          }
23
24          Console.WriteLine(Thread.CurrentThread.Name + " Sleeping");
25          Thread.Sleep(10);
26          Console.WriteLine(Thread.CurrentThread.Name + " Pulse waiting threads...");
27          Monitor.Pulse(_list);
28
29          for(int i=0; i<ITEM_COUNT; i++){
30            _list.Add(_random.Next(500));
31          }
32        }
33       }catch(Exception e){
34          Console.WriteLine(e);
35      }finally{
36        if(lockTaken){
37         Monitor.Exit(_list);
38         Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
39       }
40      }
41       Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
42    }
43
44    public void ReaderMethod(){
45       Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
46      bool lockTaken = false;
47      try{
48      while(!lockTaken){
49        Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
50        Monitor.Enter(_list, ref lockTaken);
51        if(lockTaken){
52          Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
53          if(_list.Count == 0){
54            Console.WriteLine(Thread.CurrentThread.Name + " List is currently empty. Releasing the lock.");
55            Monitor.Wait(_list);
56          }
57          foreach(int i in _list){
58            Console.Write(i + " ");
59            Console.Write(Thread.CurrentThread.Name + " Sleeping  ");
60            Thread.Sleep(10);
61          }
62        }
63      }
64     }catch(Exception e){
65       Console.WriteLine(e);
66     }finally{
67       if(lockTaken){
68        Monitor.Exit(_list);
69         Console.WriteLine(Thread.CurrentThread.Name + " Lock relinquished");
70       }
71     }
72      Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
73    }
74
75
76    public static void Main(){
77      MonitorLockTakenDemo mltd = new MonitorLockTakenDemo();
78      Thread t1 = new Thread(mltd.InserterMethod);
79      Thread t2 = new Thread(mltd.ReaderMethod);
80      t1.Name = "Inserter Thread";
81      t2.Name = "Reader Thread";
82      t2.Start();
83      Thread.Sleep(10);
84      t1.Start();
85      t1.Join();
86      t2.Join();
87    }
88 }
```

Referring to example 14.7 — this example, while similar to the previous examples, is structured differently. It still consists of two primary threads, t1 and t2. Thread t1 is the Inserter thread and t2 is the Reader thread. However, the Main() method starts t2 first to demonstrate what happens when the ReaderMethod() finds the _list empty.

Referring to the ReaderMethod() which begins on line 44 — a local variable named lockTaken is declared and initialized to false on line 46. The `try` block begins on the next line which includes a `while` loop that checks the value of lockTaken. If lockTaken is false, a call to the overloaded Monitor.Enter() method is made passing in a refer-

ence to the _list as the first argument and the lockTaken variable passed in using the ref keyword as the second argument. **If a lock already exists on _list, the call to Monitor.Enter() will block until the lock is released and acquired.** When the lock is acquired, the lockTaken variable is set to true and the `if` statement on line 51 is entered. The `if` statement on line 53 checks the value of _list.Count and if it finds the list empty it releases the lock with a call to Monitor.Wait(_list). **The call to Monitor.Wait() blocks until the lock is again acquired.** When the lock is reacquired, the `foreach` statement on line 57 executes and enumerates through the collection printing the items to the console, making a call to Thread.Sleep(10) during each iteration.

Referring to the InserterMethod() on line 12 — a local variable named lockTaken is declared and initialized to false on line 14. On line 17 the overloaded version of Monitor.Enter() is called. When the lock becomes available, the InserterMethod() will start to insert integers into the _list. After the first `for` statement the thread is put to sleep with a call to Thread.Sleep(10) followed by a call to Monitor.Pulse(_list) which signals threads waiting to obtain a lock on the _list object to wake up and try to obtain the lock.

In the Main() method which begins on line 76, thread t2 is started first followed by a call to Thread.Sleep(10), which puts the Main thread to sleep, giving a chance for the t2 thread to get going before calling t1.Start(). Figure 14-9 shows the results of running this program.

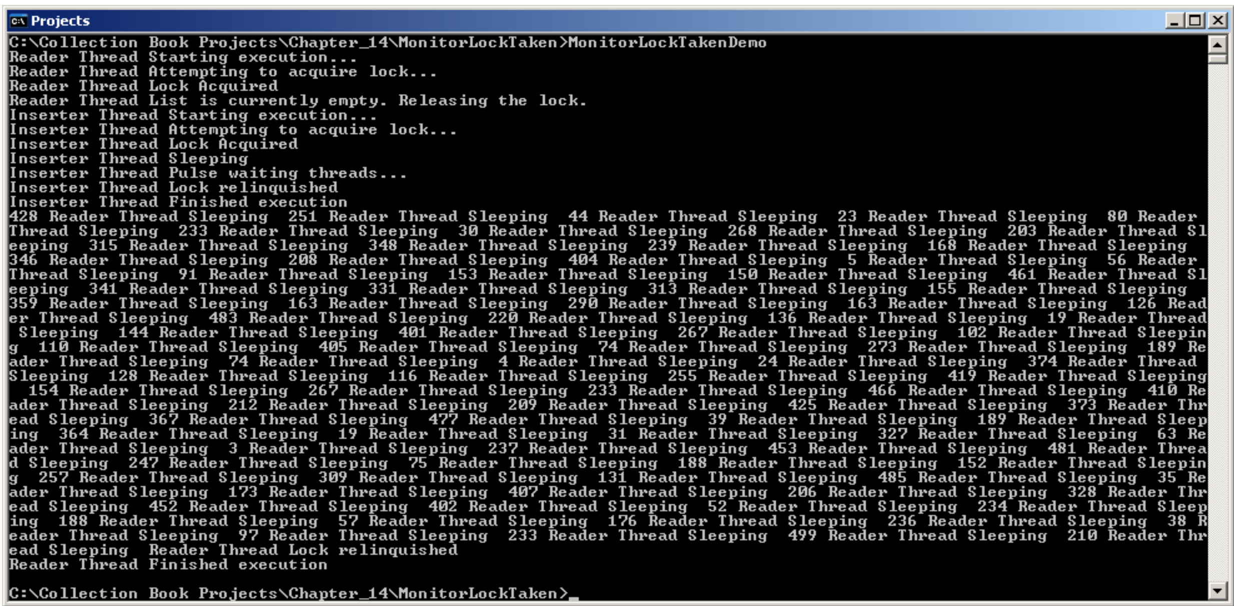


Figure 14-9: Results of Running Example 14.7

Non-Blocking Monitor.TryEnter()

The Monitor.TryEnter() method is a non-blocking method, which means that regardless of whether or not the lock is acquired, the method will immediately return. This method is also overloaded and the use of the two-argument version is recommend going forward. Example 14.8 demonstrates the use of the Monitor.TryEnter() method.

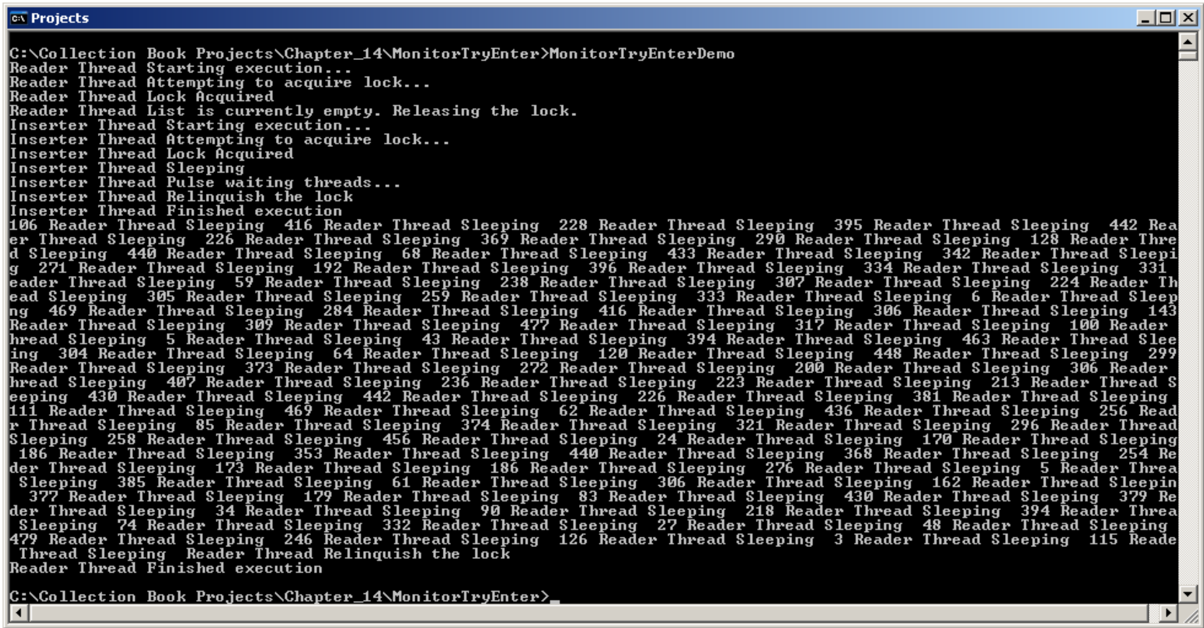
14.8 MonitorTryEnterDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;

public class MonitorTryEnterDemo {

   private List<int>  _list = new List<int>();
   private Random _random = new Random();
   private const int ITEM_COUNT = 50;


   public void InserterMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
     bool lockTaken = false;
     try{
       Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
```



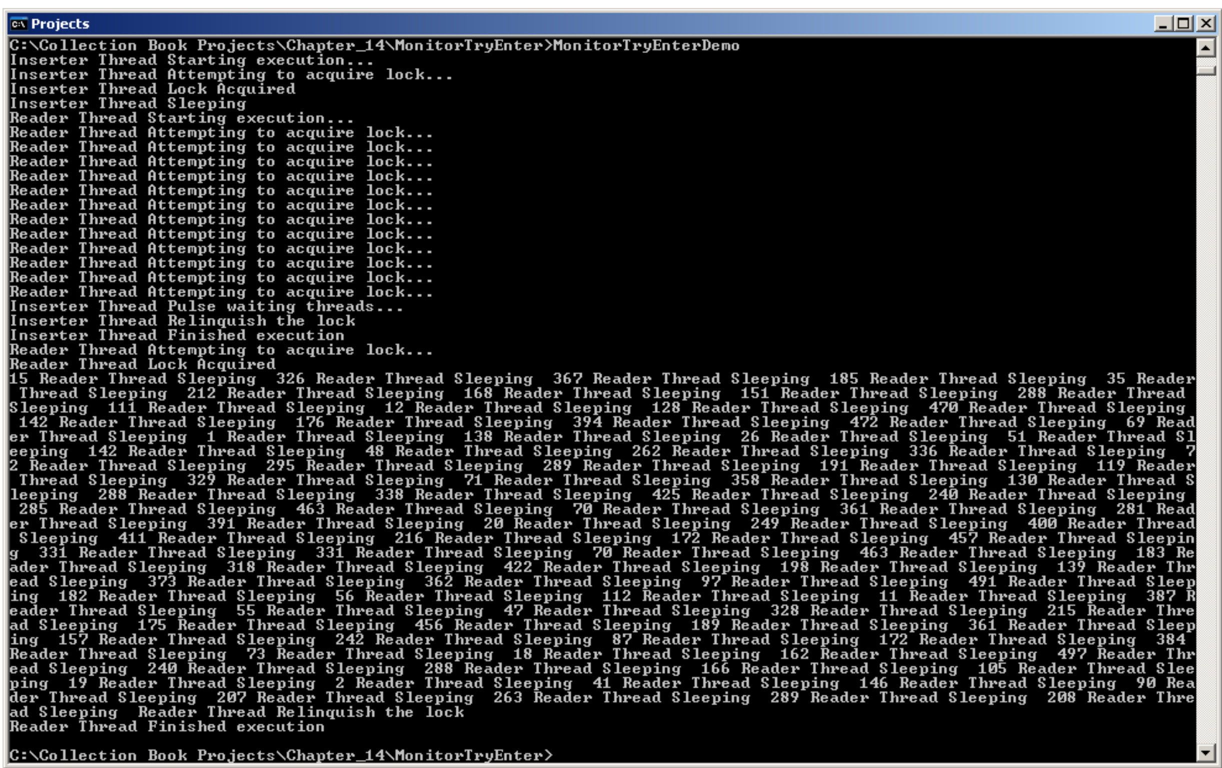
```
        Monitor.TryEnter(_list, ref lockTaken);
        if(lockTaken){
          Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
          for(int i=0; i<ITEM_COUNT; i++){
            _list.Add(_random.Next(500));
          }

          Console.WriteLine(Thread.CurrentThread.Name + " Sleeping");
          Thread.Sleep(10);
           Console.WriteLine(Thread.CurrentThread.Name + " Pulse waiting threads...");
          Monitor.Pulse(_list);

          for(int i=0; i<ITEM_COUNT; i++){
            _list.Add(_random.Next(500));
          }
        }
       }catch(Exception e){
          Console.WriteLine(e);
      }finally{
        if(lockTaken){
         Monitor.Exit(_list);
          Console.WriteLine(Thread.CurrentThread.Name + " Relinquish the lock");
       }
      }
      Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
    }

    public void ReaderMethod(){
      Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
      bool lockTaken = false;
      try{
      while(!lockTaken){
        Console.WriteLine(Thread.CurrentThread.Name + " Attempting to acquire lock...");
        Monitor.TryEnter(_list, ref lockTaken);
        if(lockTaken){
          Console.WriteLine(Thread.CurrentThread.Name + " Lock Acquired");
          if(_list.Count == 0){
            Console.WriteLine(Thread.CurrentThread.Name + " List is currently empty. Releasing the lock.");
            Monitor.Wait(_list);
          }
          foreach(int i in _list){
            Console.Write(i + " ");
            Console.Write(Thread.CurrentThread.Name + " Sleeping  ");
            Thread.Sleep(10);
          }
        }
      }
     }catch(Exception e){
       Console.WriteLine(e);
     }finally{
       if(lockTaken){
        Monitor.Exit(_list);
        Console.WriteLine(Thread.CurrentThread.Name + " Relinquish the lock");
       }
     }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
    }


    public static void Main(){
      MonitorTryEnterDemo mted = new MonitorTryEnterDemo();
      Thread t1 = new Thread(mted.InserterMethod);
      Thread t2 = new Thread(mted.ReaderMethod);
      t1.Name = "Inserter Thread";
      t2.Name = "Reader Thread";
      t2.Start();
      Thread.Sleep(10);
      t1.Start();
      t1.Join();
      t2.Join();
    }
 }
```

Referring to example 14.8 — this program is similar to the previous example, only the Monitor.TryEnter() method is used in place of the Monitor.Enter() method. Note that even though I'm starting thread t2 first, there is no guarantee it will start first. (And this applies to the previous example as well.) Figures 14-10 and 14-11 show two possible outcomes from running this program repeatedly.



```
Projects
C:\Collection Book Projects\Chapter_14\MonitorTryEnter>MonitorTryEnterDemo
Reader Thread Starting execution...
Reader Thread Attempting to acquire lock...
Reader Thread Lock Acquired
Reader Thread List is currently empty. Releasing the lock.
Inserter Thread Starting execution...
Inserter Thread Attempting to acquire lock...
Inserter Thread Lock Acquired
Inserter Thread Sleeping
Inserter Thread Pulse waiting threads...
Inserter Thread Relinquish the lock
Inserter Thread Finished execution
106 Reader Thread Sleeping  416 Reader Thread Sleeping  228 Reader Thread Sleeping  395 Reader Thread Sleeping  442 Rea
er Thread Sleeping  226 Reader Thread Sleeping  369 Reader Thread Sleeping  290 Reader Thread Sleeping  128 Reader Thre
d Sleeping  440 Reader Thread Sleeping  68 Reader Thread Sleeping  433 Reader Thread Sleeping  342 Reader Thread Sleepi
g  271 Reader Thread Sleeping  192 Reader Thread Sleeping  396 Reader Thread Sleeping  334 Reader Thread Sleeping  331
eader Thread Sleeping  59 Reader Thread Sleeping  238 Reader Thread Sleeping  307 Reader Thread Sleeping  224 Reader Th
ead Sleeping  305 Reader Thread Sleeping  259 Reader Thread Sleeping  333 Reader Thread Sleeping  6 Reader Thread Sleep
ng  469 Reader Thread Sleeping  284 Reader Thread Sleeping  416 Reader Thread Sleeping  306 Reader Thread Sleeping  143
Reader Thread Sleeping  309 Reader Thread Sleeping  477 Reader Thread Sleeping  317 Reader Thread Sleeping  100 Reader
hread Sleeping  5 Reader Thread Sleeping  43 Reader Thread Sleeping  394 Reader Thread Sleeping  463 Reader Thread Slee
ing  304 Reader Thread Sleeping  64 Reader Thread Sleeping  120 Reader Thread Sleeping  448 Reader Thread Sleeping  299
Reader Thread Sleeping  373 Reader Thread Sleeping  272 Reader Thread Sleeping  200 Reader Thread Sleeping  306 Reader
hread Sleeping  407 Reader Thread Sleeping  236 Reader Thread Sleeping  223 Reader Thread Sleeping  213 Reader Thread S
eeping  430 Reader Thread Sleeping  442 Reader Thread Sleeping  226 Reader Thread Sleeping  381 Reader Thread Sleeping
111 Reader Thread Sleeping  469 Reader Thread Sleeping  62 Reader Thread Sleeping  436 Reader Thread Sleeping  256 Read
r Thread Sleeping  85 Reader Thread Sleeping  374 Reader Thread Sleeping  321 Reader Thread Sleeping  296 Reader Thread
Sleeping  258 Reader Thread Sleeping  456 Reader Thread Sleeping  24 Reader Thread Sleeping  170 Reader Thread Sleeping
 186 Reader Thread Sleeping  353 Reader Thread Sleeping  440 Reader Thread Sleeping  368 Reader Thread Sleeping  254 Re
der Thread Sleeping  173 Reader Thread Sleeping  186 Reader Thread Sleeping  276 Reader Thread Sleeping  5 Reader Threa
 Sleeping  385 Reader Thread Sleeping  61 Reader Thread Sleeping  306 Reader Thread Sleeping  162 Reader Thread Sleepin
  377 Reader Thread Sleeping  179 Reader Thread Sleeping  83 Reader Thread Sleeping  430 Reader Thread Sleeping  379 Re
der Thread Sleeping  34 Reader Thread Sleeping  90 Reader Thread Sleeping  218 Reader Thread Sleeping  394 Reader Threa
 Sleeping  74 Reader Thread Sleeping  332 Reader Thread Sleeping  27 Reader Thread Sleeping  48 Reader Thread Sleeping
479 Reader Thread Sleeping  246 Reader Thread Sleeping  126 Reader Thread Sleeping  3 Reader Thread Sleeping  115 Reade
 Thread Sleeping  Reader Thread Relinquish the lock
Reader Thread Finished execution

C:\Collection Book Projects\Chapter_14\MonitorTryEnter>_
```

Figure 14-10: Results of Running Example 14.8



```
Projects
C:\Collection Book Projects\Chapter_14\MonitorTryEnter>MonitorTryEnterDemo
Inserter Thread Starting execution...
Inserter Thread Attempting to acquire lock...
Inserter Thread Lock Acquired
Inserter Thread Sleeping
Reader Thread Starting execution...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Reader Thread Attempting to acquire lock...
Inserter Thread Pulse waiting threads...
Inserter Thread Relinquish the lock
Inserter Thread Finished execution
Reader Thread Attempting to acquire lock...
Reader Thread Lock Acquired
15 Reader Thread Sleeping  326 Reader Thread Sleeping  367 Reader Thread Sleeping  185 Reader Thread Sleeping  35 Reader
 Thread Sleeping  212 Reader Thread Sleeping  168 Reader Thread Sleeping  151 Reader Thread Sleeping  288 Reader Thread
Sleeping  111 Reader Thread Sleeping  12 Reader Thread Sleeping  128 Reader Thread Sleeping  470 Reader Thread Sleeping
 142 Reader Thread Sleeping  176 Reader Thread Sleeping  394 Reader Thread Sleeping  472 Reader Thread Sleeping  69 Read
er Thread Sleeping  1 Reader Thread Sleeping  138 Reader Thread Sleeping  26 Reader Thread Sleeping  51 Reader Thread Sl
eeping  142 Reader Thread Sleeping  48 Reader Thread Sleeping  262 Reader Thread Sleeping  336 Reader Thread Sleeping  7
2 Reader Thread Sleeping  295 Reader Thread Sleeping  289 Reader Thread Sleeping  191 Reader Thread Sleeping  119 Reader
 Thread Sleeping  329 Reader Thread Sleeping  71 Reader Thread Sleeping  358 Reader Thread Sleeping  130 Reader Thread S
leeping  288 Reader Thread Sleeping  338 Reader Thread Sleeping  425 Reader Thread Sleeping  240 Reader Thread Sleeping
 285 Reader Thread Sleeping  463 Reader Thread Sleeping  70 Reader Thread Sleeping  361 Reader Thread Sleeping  281 Read
er Thread Sleeping  391 Reader Thread Sleeping  20 Reader Thread Sleeping  249 Reader Thread Sleeping  400 Reader Thread
 Sleeping  411 Reader Thread Sleeping  216 Reader Thread Sleeping  172 Reader Thread Sleeping  457 Reader Thread Sleepin
g  331 Reader Thread Sleeping  331 Reader Thread Sleeping  70 Reader Thread Sleeping  463 Reader Thread Sleeping  183 Re
ader Thread Sleeping  318 Reader Thread Sleeping  422 Reader Thread Sleeping  198 Reader Thread Sleeping  139 Reader Thr
ead Sleeping  373 Reader Thread Sleeping  362 Reader Thread Sleeping  97 Reader Thread Sleeping  491 Reader Thread Sleep
ing  182 Reader Thread Sleeping  56 Reader Thread Sleeping  112 Reader Thread Sleeping  11 Reader Thread Sleeping  387 R
eader Thread Sleeping  55 Reader Thread Sleeping  47 Reader Thread Sleeping  328 Reader Thread Sleeping  215 Reader Thre
ad Sleeping  175 Reader Thread Sleeping  456 Reader Thread Sleeping  189 Reader Thread Sleeping  361 Reader Thread Sleep
ing  157 Reader Thread Sleeping  242 Reader Thread Sleeping  87 Reader Thread Sleeping  172 Reader Thread Sleeping  384
Reader Thread Sleeping  73 Reader Thread Sleeping  18 Reader Thread Sleeping  162 Reader Thread Sleeping  497 Reader Thr
ead Sleeping  240 Reader Thread Sleeping  288 Reader Thread Sleeping  166 Reader Thread Sleeping  105 Reader Thread Slee
ping  19 Reader Thread Sleeping  2 Reader Thread Sleeping  41 Reader Thread Sleeping  146 Reader Thread Sleeping  90 Rea
der Thread Sleeping  207 Reader Thread Sleeping  263 Reader Thread Sleeping  289 Reader Thread Sleeping  208 Reader Thre
ad Sleeping  Reader Thread Relinquish the lock
Reader Thread Finished execution

C:\Collection Book Projects\Chapter_14\MonitorTryEnter>
```

Figure 14-11: Another Possible Result of Running Example 14.8

Quick Review

The static Monitor class allows you to implement fine grained thread synchronization. You must be sure that for each call to Monitor.Enter(_lockObject) is followed by a call to Monitor.Exit(_lockObject). Failure to do so may

result in deadlock as waiting threads will never acquire an unreleased lock. The critical code section begins with a call to Monitor.Enter(). Place the call to Monitor.Exit() in the body of the `finally` clause of a `try/catch/finally` block. **The Monitor.Enter() method blocks until it acquires the lock.** The Monitor.Enter() method is overloaded. Favor the use of the two-argument version of Monitor.Enter() going forward.

The Monitor.TryEnter() method is a non-blocking method that returns immediately after it's called regardless of whether or not the lock is acquired. You must take this immediate return behavior into account in your code. Use the overloaded two-argument version of the Monitor.TryEnter() method to test whether or not the lock was acquired.

If a thread needs to give up the lock because it has nothing to do, call the Monitor.Wait() method. To signal waiting threads of a change in lock status, call the Monitor.Pulse() method to move the next waiting thread into the ready queue.

Synchronizing Entire Methods

If you're using the C# `lock` keyword to synchronize significant portions of a method's body, you can alternatively tag the entire method as being synchronized using the [MethodImpl(MethodImplOptions.Synchronized)] attribute. It's easy to use. Simply apply the attribute to each method you want to synchronize.

14.9 SynchronizedMethodDemo.cs

```
using System;
using System.Threading;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

public class SynchronizedMethodDemo {

   private List<int> _list = new List<int>();
   private Random _random = new Random();
   private const int ITEM_COUNT = 50;

   [MethodImpl(MethodImplOptions.Synchronized)]
   public void InserterMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution...");
     try {
       for(int i=0; i<ITEM_COUNT; i++){
         _list.Add(_random.Next(500));
       }

       Thread.Sleep(10);

       for(int i=0; i<ITEM_COUNT; i++){
         _list.Add(_random.Next(500));
       }
     }catch(Exception e){
       Console.WriteLine(e);
     }
     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }

   [MethodImpl(MethodImplOptions.Synchronized)]
   public void ReaderMethod(){
     Console.WriteLine(Thread.CurrentThread.Name + " Starting execution");

     try{
       foreach(int i in _list){
         Console.Write(i + " ");
         Thread.Sleep(10);
       }
     }catch(Exception e){
       Console.WriteLine(e);
     }

     Console.WriteLine(Thread.CurrentThread.Name + " Finished execution");
   }


   public static void Main(){
     SynchronizedMethodDemo smd = new SynchronizedMethodDemo();
     Thread t1 = new Thread(smd.InserterMethod);
     Thread t2 = new Thread(smd.ReaderMethod);
     t1.Name = "Inserter Thread";
     t2.Name = "Reader Thread";
```



```
      t1.Start();
      t2.Start();
      t1.Join();
      t2.Join();
    }
}
```

Referring to example 14.9 — the [MethodImpl(MethodImplOptions.Synchronized)] attribute is applied to both thread methods. The use of the [MethodImpl(MethodImplOptions.Synchronized)] attribute is essentially applying the Monitor.Enter()/Monitor.Exit() thread synchronization mechanism to the entire body of the method, locking on the instance (i.e., Monitor.Enter(this)/Monitor.Exit(this)). Figure 14-11 shows the results of running this program.

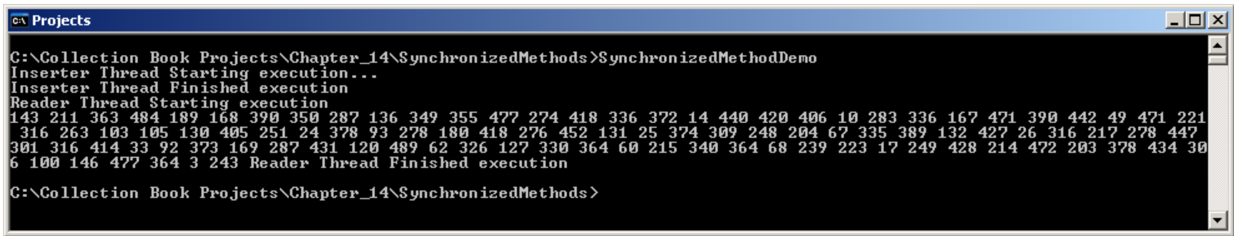


Figure 14-12: Results of Running Example 14.9

Quick Review

Use the [MethodImpl(MethodImplOptions.Synchronized)] attribute to synchronize entire methods. However, I recommend using this attribute sparingly. Generally speaking, the finer grained you can make your thread synchronization scheme, the better off you'll be.

Synchronized Collections In The System.Collections.Generic Namespace

The System.Collections.Generic namespace contains three "synchronized" collections named: SynchronizedCollection<T>, SynchronizedKeyedCollection<T>, and SynchronizedReadOnlyCollection<T>.

I put quotes around the word "synchronized" because even though these collections start with the word Synchronized, and the .NET documentation describes each class as a "...thread-safe collection...", the documentation also says a little further down the page "Any instance members are not guaranteed to be thead safe."

So, what's so special about these collections? Well, nothing really, except that each provides a SyncRoot property that can be set via the constructor. If the default constructor is used, the SyncRoot property returns a reference to a default Object instance.

I will leave it to you to explore the use of these synchronized collections as you see fit.

Thread Synchronization – Recommendations For Usage

Thread synchronization in any form is a cooperative affair. When locking on an object, lock on the same object, otherwise the threads are synchronized on different objects, which means multiple threads might gain access to shared resources you assumed were protected. Also, lock on private field objects. In the chapter examples I locked on the list itself (e.g., _list). In a programming team environment you'll want it understood between all members upon what object within individual classes to synchronize. You may decide to define a private member object field within a class for the sole purpose of locking.

Use the C# `lock` keyword for convenience and if you don't need finer-grained thread synchronization control. You can, however, use the `lock` keyword in conjunction with the Monitor.Wait() and Monitor.Pulse() methods.

The single-argument Monitor.Enter() method is obsolete as of .NET 4.0. Going forward favor the use of the overloaded two-argument version which uses a boolean value to indicate whether or not the lock has been taken.

Other than that, this chapter has only presented and demonstrated a small sampling of the thread synchronization mechanisms available to you in the .NET platform. However, you can accomplish a lot with thread synchronization using the various methods of the Monitor class.

With regards to collections, the important thing to remember is that an exception will be thrown when attempting to enumerate a collection that is being simultaneously modified by another thread.

Thread Synchronization Usage Table

Table 14-1 lists and summarizes the thread synchronization mechanisms presented in this chapter.

Synchronization Primitive	Category	Usage	Comments

C# `lock` keyword	locking	`lock(_lockObject){`
		`  //critical section`
		`}`
			Translates into Monitor.Enter() and Monitor.Exit() calls under the covers.

Monitor class
Monitor.Enter()
Monitor.Exit()
(basic usage)
	locking	`Monitor.Enter(_lockObject);`
		`try{`
		`  //critical code section`
		`}catch(Exception e){`
		`  //exception handler code`
		`}finally{`
		`  Monitor.Exit(_lockObject);`
		`}`
			Obsolete as of .NET 4.0. (Source: Compiler warning csc version 4.0.21006.1)
			If a lock already exists on _lockObject the thread blocks until the lock on _lockObject is released.

Monitor class
Monitor.Enter()
Monitor.Exit()
(overloaded method usage with lockTaken boolean argument)
	locking	`bool lockTaken = false;`
		`try{`
		`  Monitor.Enter(_lockObject,`
		`                ref lockTaken);`
		`  if(lockTaken){`
		`    // do this if lock taken`
		`  }else{`
		`    // alternative processing`
		`  }`
		`}catch(Exception e){`
		
		`}finally{`
		`  if(lockTaken){`
		`    Monitor.Exit(_lockObject);`
		`  }`
		`}`
			Preferred use as of .NET 4.0. (Source: Compiler warning csc version 4.0.21006.1)
			If a lock already exists on _lockObject the thread blocks until the lock on _lockObject is released.
			The value of the lockTaken argument is set even if an exception is thrown when attempting to acquire the lock on _lockObject.

Table 14-1: Synchronization Primitives Reference Table

Synchronization Primitive	Category	Usage	Comments

Monitor class
Monitor.Enter()
Monitor.Exit()
(Fine grain control with Monitor.Wait() and Monitor.Pulse())
	locking	public void MethodA(){
		  bool lockTaken = false;
		  try{
		    Monitor.Enter(_lockObject,
		                  ref lockTaken);
		    if(lockTaken){
		      // do this if lock taken
		      // if resource not available
		      // block until it is...
		      Monitor.Wait(_lockObject);
		      // when lock reacquired
		      // continue processing
		    }else{
		      // alternative processing
		    }
		  }catch(Exception e){
		
		  }finally{
		    if(lockTaken){
		      Monitor.Exit(_lockObject);
		    }
		  }
		} // end MethodA()
		
		public void MethodB(){
		  bool lockTaken = false;
		  try{
		    Monitor.Enter(_lockObject,
		                  ref lockTaken);
		    if(lockTaken){
		      // do this if lock taken
		      // if you want to relinquish
		      // the lock for a while...
		      Monitor.Pulse(lockObject);
		      Thread.Sleep(10);
		      // to give other threads
		      // a chance to execute
		    }else{
		      // alternative processing
		    }
		  }catch(Exception e){
		
		  }finally{
		    if(lockTaken){
		      Monitor.Exit(_lockObject);
		    }
		  }
		} // end MethodB()
			The thread that currently owns the lock on an object calls Monitor.Wait(object) to relinquish the lock and block until it can reacquire the lock.
			Another thread must make a call to Monitor.Pulse(object) to signal blocked threads that are waiting on the lock object to move to the ready queue.
			Note: This is a cooperative scheme. If one thread calls Wait() without another thread's corresponding call to Pulse() then deadlock can occur because one thread is blocked indefinitely waiting for the other thread to signal it to move to the ready queue.

Table 14-1: Synchronization Primitives Reference Table

<table>
<tr><th>Synchronization Primitive</th><th>Category</th><th>Usage</th><th>Comments</th></tr>
<tr><td>Monitor class
Monitor.TryEnter()
Monitor.Exit()</td><td>locking</td><td><pre>public void MethodA(){
  bool lockTaken = false;
  try{
    Monitor.TryEnter(_lockObject,
                     ref lockTaken);
    if(lockTaken){
      // do this if lock taken
      // if resource not available
      // block until it is...
      Monitor.Wait(_lockObject);
      // when lock reacquired
      // continue processing
    }else{
      // alternative processing
    }
  }catch(Exception e){

  }finally{
    if(lockTaken){
      Monitor.Exit(_lockObject);
    }
  }
} // end MethodA()

public void MethodB(){
  bool lockTaken = false;
  try{
    Monitor.TryEnter(_lockObject,
                     ref lockTaken);
    if(lockTaken){
      // do this if lock taken
      // if you want to relinquish
      // the lock for a while...
      Monitor.Pulse(lockObject);
      Thread.Sleep(10);
      // to give other threads
      // a chance to execute
    }else{
      // alternative processing
    }
  }catch(Exception e){

  }finally{
    if(lockTaken){
      Monitor.Exit(_lockObject);
    }
  }
} // end MethodB()</pre></td><td>The Monitor.TryEnter() method does not block. It returns immediately</td></tr>
</table>

Table 14-1: Synchronization Primitives Reference Table

Synchronization Primitive	Category	Usage	Comments

MethodImplOptions. Synchronized Attribute	Contextual	`[MethodImpl(MethodImplOptions.Synchronized)]`
		`public void MethodName(){`
		`  // the entire method is synchronized`
		`}`
			Synchronizes the entire method.

Table 14-1: Synchronization Primitives Reference Table

Summary

The need for thread synchronization arises when multiple threads of execution may access shared resources or shared code segments, which, if unsynchronized, would destabilize the code or leave the code in an invalid state. The .NET framework and the C# language provide various thread synchronization primitives and strategies that enable you to synchronize thread access to critical code segments.

The C# `lock` keyword is the easiest way to protect critical code segments. Use the C# lock keyword to obtain a "lock" on an object. Place the code you want to protect within the body of the `lock` statement. **Recommendation: Lock on private field objects only.** Do not lock on the current instance (i.e. this). **Warning: Do not lock on value objects.** Value object are boxed into objects when used in a `lock` statement. Thus, multiple threads "locking" on the same value object will actually be acquiring locks on different objects.

Collection classes in the System.Collections namespace come equipped with the SyncRoot and IsSynchronized properties. These old-school collections also provided a static Synchronized() method which is used to transform an ordinary collection into a synchronized collection. And while individual collection methods may be synchronized, it was still not thread safe to enumerate over a collection. While you can still write good-quality thread-safe code using the SyncRoot property along with the `lock` keyword or the Monitor class, the use of these old-school properties, along with the Synchronized() method is best avoided. Besides, unless you find yourself maintaining legacy C# code, you should be favoring the use of the generic collection classes.

The static Monitor class allows you to implement fine grained thread synchronization. You must be sure that for each call to Monitor.Enter(_lockObject) is followed by a call to Monitor.Exit(_lockObject). Failure to do so may result in deadlock as waiting threads will never acquire an unreleased lock. The critical code section begins with a call to Monitor.Enter(). Place the call to Monitor.Exit() in the body of the `finally` clause of a `try/catch/finally` block. **The Monitor.Enter() method blocks until it acquires the lock.** The Monitor.Enter() method is overloaded. Favor the use of the two-argument version of Monitor.Enter() going forward.

The Monitor.TryEnter() method is a non-blocking method that returns immediately after its called regardless of whether or not the lock is acquired. You must take this immediate return behavior into account in your code. Use the overloaded two-argument version of the Monitor.TryEnter() method to test whether or not the lock was acquired.

If a thread needs to give up the lock because it has nothing to do, call the Monitor.Wait() method. To signal waiting threads of a change in lock status, call the Monitor.Pulse() method to move the next waiting thread into the ready queue.

Use the [MethodImpl(MethodImplOptions.Synchronized)] attribute to synchronize entire methods. However, I recommend using this attribute sparingly. Generally speaking, the finer grained you can make your thread synchronization scheme, the better off you'll be.

Reference

Microsoft Developer Network (MSDN) *.NET Framework 3.0, 3.5, and 4.0 Reference Documentation* [www.msdn.com]

Microsoft *Shared Source Common Language Infrastructure 2.0* Release (SSCLI 2.0)(Codename: Rotor)[http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en]

Notes

Chapter 15

Contax T

Carnival Ride

Events And Event Processing

Learning Objectives

- *Describe the .NET event handling process*
- *Create custom events*
- *Understand the role of delegate types*
- *Create delegate types that define event handler methods signatures*
- *Create events using delegate types*
- *Create an event publisher*
- *Create an event subscriber*

Introduction

Graphical User Interface (GUI) components are not the only type of objects that can have event members. (*Refer to chapter 12, C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*) Indeed, you can add events to the classes you design, and that's the subject of this chapter.

To add custom events to your programs, you'll need to know a little something about the following topics: 1) how to use the *delegate* keyword to declare new delegate types, 2) how to use the *event* keyword to declare new event members using delegate types, 3) how to create a class that conveys event data between an *event publisher* and an *event subscriber*, 4) how to create an event publisher, 5) how to create an event subscriber, 6) how to create *event handler methods*, and 7) how to *register event handlers* with a particular event.

The information and programming techniques you will learn in this chapter will enable you to understand how event-driven collections work beneath the covers and prepare you for Chapter 16 — Events and Collections.

C# Event Processing Model: An Overview

C# is a modern programming language supported by an extensive Application Programming Interface (API) referred to as the .NET Framework. One normally thinks of events as being generated exclusively by GUI components, like buttons for example, but any object can generate an event if it's programmed to do so.

You need two logical components to implement the event processing model: 1) an event producer (or *publisher*), and 2) an event consumer (or *subscriber*). Each component has certain responsibilities. Consider the following diagram:

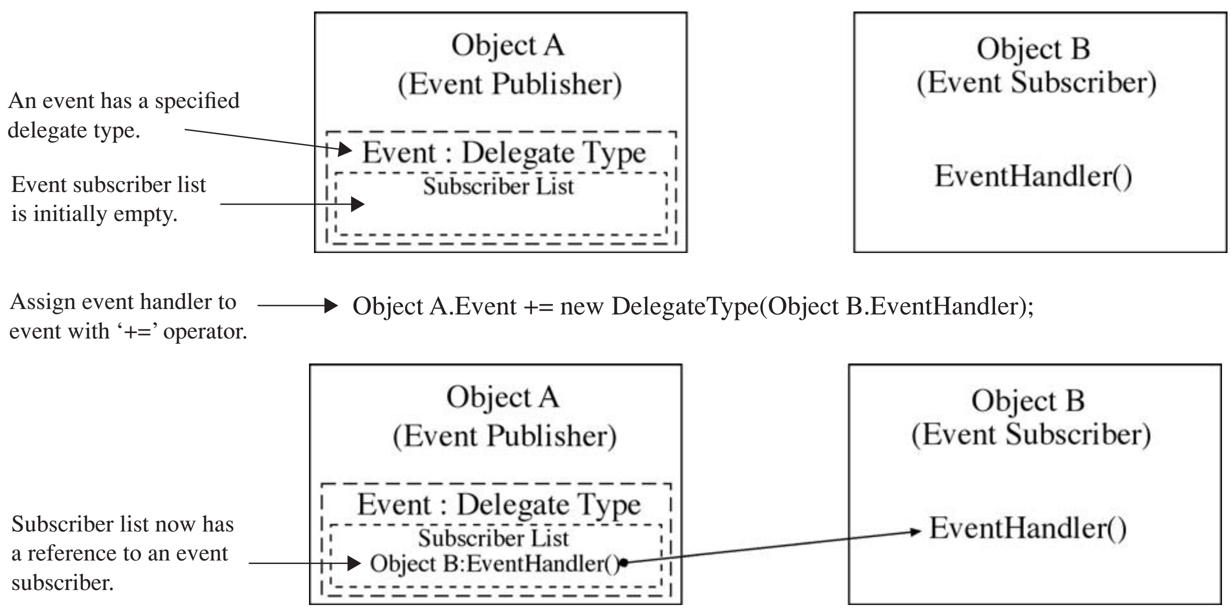


Figure 15-1: Event Publisher and Subscriber

Referring to figure 15-1 — each event in Object A has a specified delegate type. The delegate type specifies the authorized method signature for event handler methods. However, the delegate does more than just specify a method signature; a delegate object maintains a list of event subscribers in the form of references to event handler methods. These references can point to an object and one of its instance methods or to a static method. The event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. The EventHandler() method defined in Object B must conform to the method signature specified by the event's delegate type. If you attempt to use an event handler method that does not conform to the delegate type's method signature, you will receive a compiler error. Let's now substitute some familiar names for Object A and Object B. Figure 15-2 offers a revised diagram.

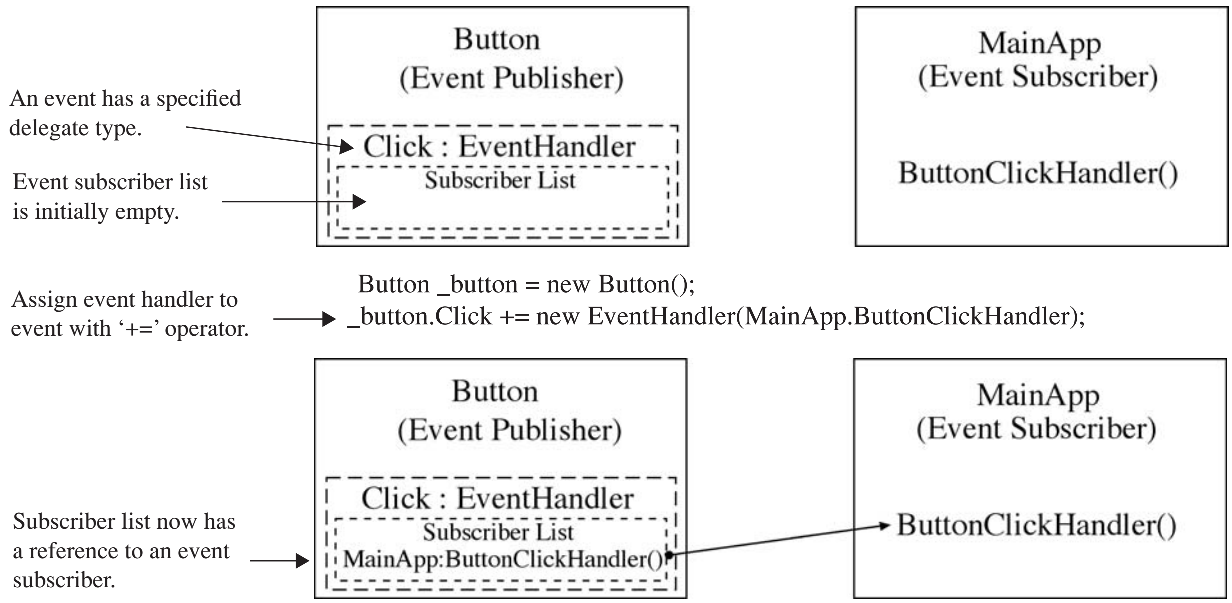


Figure 15-2: Event Publisher and Subscriber

Referring to figure 15-2 — a button's Click event has an EventHandler delegate type. Event handler methods assigned to a button's Click event must have the following signature:

```
void MethodName(object sender, EventArgs e)
```

The method's name can be pretty much anything you want it to be, but it must declare two parameters, the first of type *object*, and the second of type *EventArgs*, and it must return void. The names of the parameters can be anything you want as well, but the names *sender* and *e* work just fine.

When the button's Click event is fired, the Click event's delegate instance calls each registered subscriber's event handler method in the order it appears in the event subscriber list. This is all handled behind the scenes as you will soon see. In the case of a button and other controls, there exists an internal method that is called when a Click event occurs that kicks off the subscriber notification process. Remember that when talking about GUI components, a mouse click results in the generation of a message that is ultimately routed to the window in which the mouse click occurred. This message is translated into a Click event. When writing custom events, you can intercept messages and translate them into events or write a method that generates events out of thin air or in response to some other stimulus.

Quick Review

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

Custom Events Example: Minute Tick

The best way to get your head around custom events is to study an example application. This section presents a short program that implements custom events. The Minute Tick application consists of five source files, four of which appear in the Unified Modeling Language (UML) class diagram shown in figure 15-3.

Referring to figure 15-3 — both the Publisher and Subscriber classes depend on the MinuteEventArgs class and the ElapsedMinuteEventHandler delegate. The Publisher class contains a MinuteTick event, which is of type

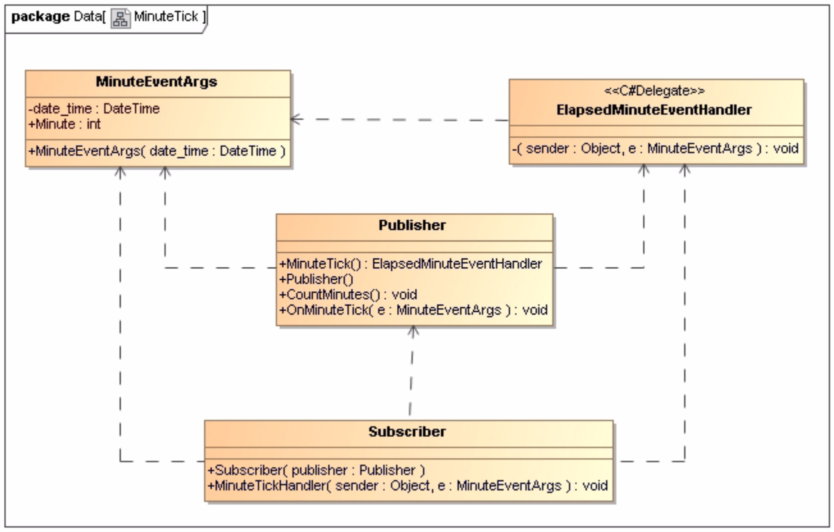


Figure 15-3: Minute Tick UML Class Diagram

ElapsedMinuteEventHandler. The ElapsedMinuteEventHandler delegate depends on the MinuteEventArgs class because it is the type of one of its parameters, as is shown in the diagram.

The complete Minute Tick application source code is given in examples 15.1 through 15.5.

15.1 MinuteEventArgs.cs

```
using System;

public class MinuteEventArgs : EventArgs {
  private DateTime date_time;

  public MinuteEventArgs(DateTime date_time){
    this.date_time = date_time;
  }

  public int Minute {
    get {  return date_time.Minute; }
  }
}
```

Referring to example 15.1 — the MinuteEventArgs class extends the EventArgs class and adds a private field named *date_time* and one public read-only property named *Minute,* which simply returns the value of the Minute property of the DateTime object.

15.2 ElapsedMinuteEventHandler.cs

```
using System;

public delegate void ElapsedMinuteEventHandler(Object sender, MinuteEventArgs e);
```

Referring to example 15.2 — the ElapsedMinuteEventHandler delegate specifies a method signature that returns `void` and takes two parameters, the first one an object, and the second one of type MinuteEventArgs.

15.3 Publisher.cs

```
using System;

public class Publisher {

   public event ElapsedMinuteEventHandler MinuteTick;


   public Publisher(){
     Console.WriteLine("Publisher Created");
   }

   public void CountMinutes(){
     int current_minute = DateTime.Now.Minute;
     while(true){
       if(current_minute != DateTime.Now.Minute){
      Console.WriteLine("Publisher: {0}", DateTime.Now.Minute);
      OnMinuteTick(new MinuteEventArgs(DateTime.Now));
      current_minute = DateTime.Now.Minute;
```

```
19          }//end if
20        }  // end while
21      }  // end CountMinutes method
22
23      public void OnMinuteTick(MinuteEventArgs e){
24         if(MinuteTick != null){
25            MinuteTick(this, e);
26          }
27      }// end OnMinuteTick method
28   }  // end Publisher class definition
```

Referring to example 15.3 — the Publisher class defines an event named *MinuteTick*. Notice that the MinuteTick event is of type ElapsedMinuteEventHandler. The CountMinutes() method that starts on line 12 contains a `while` loop that repeats forever and continuously compares the values of the current_minute with DateTime.Now.Minute. As soon as a change is detected in the two values, a brief message is written to the console followed by a call to the publisher's OnMinuteTick() method on line 17. Notice that when this method is called, a new MinuteEventArgs object is created and used as an argument to the method call. The OnMinuteTick() method definition begins on line 23. It takes the MinuteEventArgs parameter and passes it on to a call to the MinuteTick event. Note on line 24 how the `if` statement checks to see if the MinuteTick reference is null. It will be null if no event handler methods have been registered with the event.

15.4 Subscriber.cs

```
1    using System;
2
3    public class Subscriber {
4
5      public Subscriber(Publisher publisher){
6        publisher.MinuteTick += new ElapsedMinuteEventHandler(this.MinuteTickHandler);
7        Console.WriteLine("Subscriber Created");
8      }
9
10     public void MinuteTickHandler(Object sender, MinuteEventArgs e){
11       Console.WriteLine("Subscriber Handler Method: {0}", e.Minute);
12     }
13    }  // end Subscriber class definition
```

Referring to example 15.4 — the Subscriber class declares an event handler method on line 10 named *MinuteTickHandler()*. The MinuteTickHandler() method defines two arguments of the types required by the ElapsedMinuteEventHandler delegate type. The ElapsedMinuteEventHandler delegate is used on line 6 to register the subscriber's MinuteTickHandler() method with the publisher's MinuteTick event.

15.5 MainApp.cs

```
using System;

public class MainApp {
 public static void Main(){
   Console.WriteLine("Custom Events are Cool!");

   Publisher p = new Publisher();
   Subscriber s = new Subscriber(p);
   p.CountMinutes();

 }  // end main
}  //end MainApp class definition
```

Referring to example 15.5 — the MainApp class provides the Main() method. It simply creates a Publisher object and a Subscriber object, and then makes a call to the publisher's CountMinutes() method. Figure 15-4 shows the results of running this application. Note that the actual minutes displayed when the program runs depend on when you start the program.

Application started at 9 minutes past the hour. Your time outputs will reflect the time you run the program.

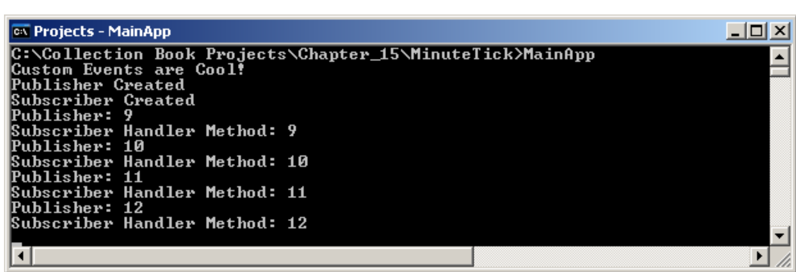


Figure 15-4: Results of Running Example 15.5

Custom Events Example: Automated Water Tank System

In this section, I want to show you how you might model a complex system using custom events. The system modeled here is a simple water tank that can be filled with water. Once the water reaches a certain level within the tank, a pump is activated and drains the tank until it again reaches a certain level. The tank contains two water level sensors. One acts as a high-level sensor and the other acts as a low-level sensor. The tank also has a pump that pumps water at a certain rate or pumping capacity. The system comprises the following classes: *Pump*, *WaterLevelEventArgs*, *WaterLevelEventHandler*, *WaterTank*, and *WaterSystemApp,* which serves as the main application class. Figure 15-5 gives the UML class diagram for the water tank system.

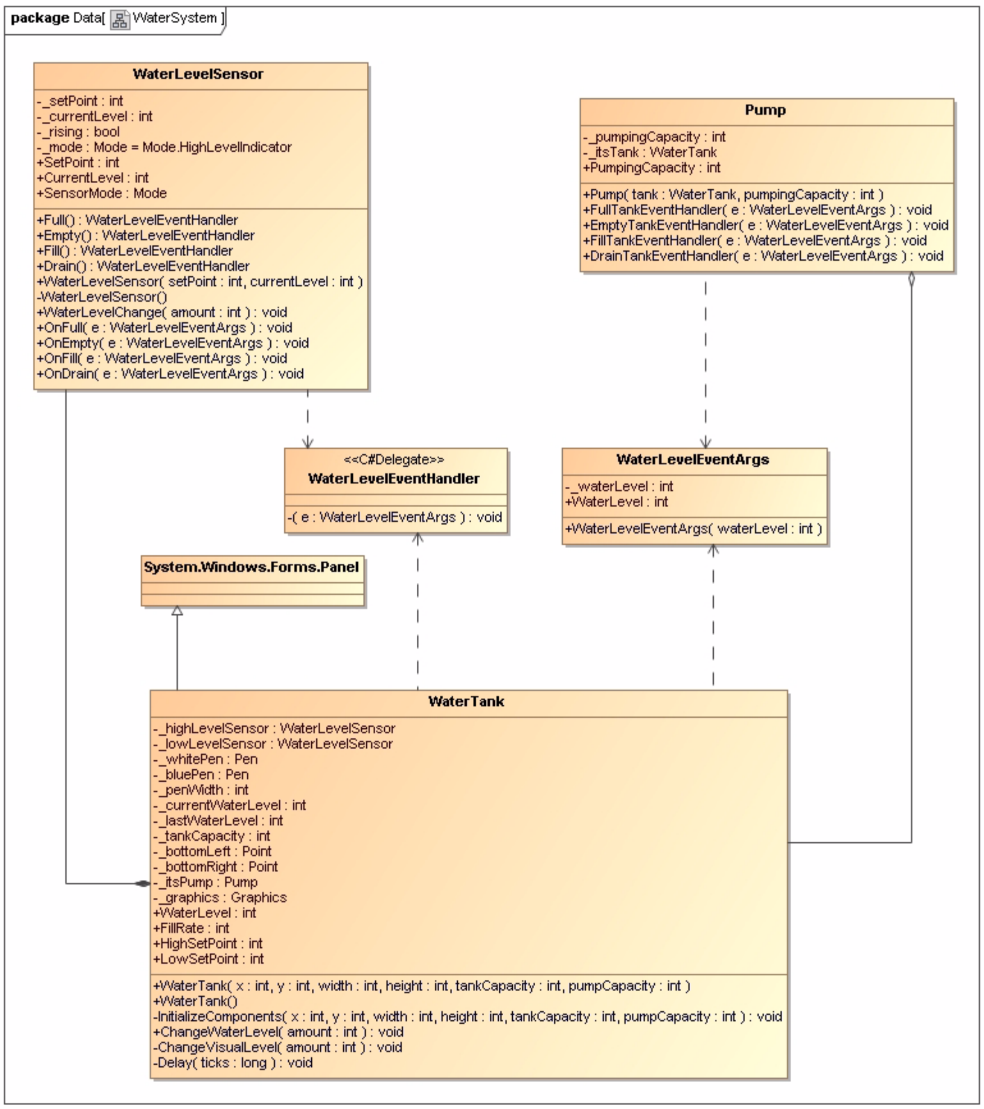


Figure 15-5: Water Tank System UML Class Diagram

Referring to figure 15-5 — The WaterTank class extends the System.Windows.Forms.Panel class. This gives the water tank a visual representation. When water is added to the tank, the panel is filled in with blue lines as the water level rises. The lines are then overdrawn with a different color as the water level recedes. The WaterLevelEventArgs class is used to pass water level information between event publisher and subscriber. In this example, the WaterLevelSensor is the publisher and the Pump is the subscriber. The WaterLevelEventHandler delegate is used to declare the WaterLevelSensor's Fill, Full, Drain, and Empty events. Let's take a look at the code.

15.6 WaterLevelEventArgs.cs

```
using System;

public class WaterLevelEventArgs : EventArgs {

  private int _waterLevel;

  public WaterLevelEventArgs(int waterLevel){
    WaterLevel = waterLevel;
 }

  public int WaterLevel {
    get { return _waterLevel; }
    set { _waterLevel = value; }
 }
}
```

Referring to example 15.6 — the WaterLevelEventArgs class contains one private integer field named *_waterLevel* and one public property named *WaterLevel*.

15.7 WaterLevelEventHandler.cs

```
using System;

public delegate void WaterLevelEventHandler(WaterLevelEventArgs e);
```

Referring to example 15.7 — the WaterLevelEventHandler delegate specifies an event-handler method signature that returns `void` and contains one parameter of type WaterLevelEventArgs.

15.8 WaterLevelSensor.cs

```
using System;

public class WaterLevelSensor {
 private int _setPoint;
 private int _currentLevel;
 private bool _rising;
 private Mode _mode = Mode.HighLevelIndicator;


 public enum Mode { HighLevelIndicator, LowLevelIndicator };
 public event WaterLevelEventHandler Full;
 public event WaterLevelEventHandler Empty;
 public event WaterLevelEventHandler Fill;
 public event WaterLevelEventHandler Drain;

 public int SetPoint {
    get { return _setPoint; }
    set { _setPoint = value; }
 }

 public int CurrentLevel {
    get { return _currentLevel; }
    set { _currentLevel = value; }
 }

 public Mode SensorMode {
    get { return _mode; }
    set { _mode = value; }
 }

 public WaterLevelSensor(int setPoint, int currentLevel){
    SetPoint = setPoint;
    CurrentLevel = currentLevel;
 }

 private WaterLevelSensor(){ }

 public void WaterLevelChange(int amount){
    int lastLevel = CurrentLevel;
    CurrentLevel += amount;
    _rising = (CurrentLevel >= lastLevel);

    switch(_mode){
     case Mode.HighLevelIndicator :
          if(_rising){
             if(CurrentLevel >= SetPoint){
             WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
             OnFull(args);
          }else{
             WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
             OnFill(args);
```



```
          }
         }
         break;

     case Mode.LowLevelIndicator :
       if(!_rising){
           if(CurrentLevel <= SetPoint){
          WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
          OnEmpty(args);
        } else{
           WaterLevelEventArgs args = new WaterLevelEventArgs(CurrentLevel);
           OnDrain(args);
        }
       }
       break;
   } // end switch
 }

 public void OnFull(WaterLevelEventArgs e){
   if(Full != null){
    Full(e);
   }
  }

 public void OnEmpty(WaterLevelEventArgs e){
    if(Empty != null){
      Empty(e);
   }
 }

 public void OnFill(WaterLevelEventArgs e){
   if(Fill != null){
     Fill(e);
   }
 }

 public void OnDrain(WaterLevelEventArgs e){
   if(Drain != null){
     Drain(e);
   }
 }
} // end WaterLevelClass definition
```

Referring to example 15.8 — the WaterLevelSensor is a primary component of the water system. Essentially, its purpose is to keep track of a tank's water level. A WaterLevelSensor object functions in one of two modes of operation as defined by the *Mode* enumeration. It can be either a *HighLevelIndicator* or a *LowLevelIndicator*. If it's operating as a HighLevelIndicator, it keeps track of rising water added via the WaterLevelChange() method. If the water level is rising, it fires the *Fill* event. When the water level reaches the set point, it fires the *Full* event.

If a WaterLevelSensor is operating in the LowLevelIndicator mode, it responds to falling water levels by firing the *Drain* event until the water reaches the low set point, at which time it fires the *Empty* event.

Each of the four events — Fill, Full, Drain, and Empty — are of type WaterLevelEventHander delegate. The Pump class, shown in the following example, defines four event handler methods that respond to each of these events.

15.9 Pump.cs

```
using System;

public class Pump {

 private int _pumpingCapacity;
 private WaterTank _itsTank;

 public int PumpingCapacity {
    get { return _pumpingCapacity; }
    set { _pumpingCapacity = value; }
 }

 public Pump(WaterTank tank, int pumpingCapacity){
    PumpingCapacity = pumpingCapacity;
    _itsTank = tank;
 }

 public void FullTankEventHandler(WaterLevelEventArgs e){
    Console.WriteLine("FullTankEventHandler: Draining the water tank!");
    _itsTank.ChangeWaterLevel(-PumpingCapacity);
 }

 public void EmptyTankEventHandler(WaterLevelEventArgs e){
    Console.Write("EmptyTankEventHandler: ");
```



```
    Console.WriteLine("Water tank has been drained! The water tank contains " +
              e.WaterLevel + " gallons!");
  }

  public void FillTankEventHandler(WaterLevelEventArgs e){
    Console.Write("FillTankEventHandler: ");
    Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
  }

  public void DrainTankEventHandler(WaterLevelEventArgs e){
    Console.Write("DrainTankEventHandler: ");
    Console.WriteLine("The water tank contains " + e.WaterLevel + " gallons!");
    _itsTank.ChangeWaterLevel(-PumpingCapacity);

  }
}
```

Referring to example 15.9 — a Pump object is created with an associated WaterTank object and a pumping capacity. Water added to the tank causes a Fill event to fire. The FillTankEventHandler() method responds by printing the value of the tank's current water level to the console. Note that the current water level is determined by reading the WaterLevelEventArgs.WaterLevel property. When the water level reaches the high level sensor's set point, the sensor fires the Full event that calls the Pump's FullTankEventHandler() method. This starts the automatic draining process by calling the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. This in turn triggers a Drain event in a WaterLevelSensor object, which calls the pump's DrainTankEventHandler() method. This results in yet another call (recursive) to the WaterTank.ChangeWaterLevel() method with a negative amount of water equal to the volume of its pumping capacity. Thus, the recursive calls to the DrainTankEventHandler() method repeat until the low-level indicator reaches its set point.

15.10 WaterTank.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class WaterTank : Panel {

 // Private fields
 private WaterLevelSensor _highLevelSensor;
 private WaterLevelSensor _lowLevelSensor;
 private Pen _whitePen;
 private Pen _bluePen;
 private int _penWidth;
 private int _currentWaterLevel;
 private int _lastWaterLevel;
 private int _tankCapacity;
 private Point _bottomLeft;
 private Point _bottomRight;
 private Pump _itsPump;
 private Graphics _graphics;

  // Constants
 private const int UPPER_LEFT_CORNER_X = 100;
 private const int UPPER_LEFT_CORNER_Y = 100;
 private const int WIDTH = 100;
 private const int HEIGHT = 500;
 private const int TANK_CAPACITY = 10000;
 private const int PUMP_CAPACITY = 1000;
 private const int ONE_PIXEL_WIDE = 1;
 private const int EMPTY = 0;

  public int WaterLevel {
    get { return _currentWaterLevel; }
 }

 public int FillRate {
    get { return _itsPump.PumpingCapacity; }
 }

 public int HighSetPoint {
    get {  return _highLevelSensor.SetPoint; }
    set {  _highLevelSensor.SetPoint = value; }
 }

 public int LowSetPoint {
    get {  return _lowLevelSensor.SetPoint; }
    set {  _lowLevelSensor.SetPoint = value; }
 }

```



```
  public WaterTank(int x, int y, int width, int height, int tankCapacity, int pumpCapacity){
     this.InitializeComponents(x, y, width, height, tankCapacity, pumpCapacity);
  }

  public WaterTank():this(UPPER_LEFT_CORNER_X, UPPER_LEFT_CORNER_Y, WIDTH, HEIGHT, TANK_CAPACITY,
                  PUMP_CAPACITY){ }

  private void InitializeComponents(int x, int y, int width, int height, int tankCapacity,
                       int pumpCapacity){

     this.Bounds = new Rectangle(x, y, width, height);
     this.BackColor = Color.White;
     this.BorderStyle = BorderStyle.Fixed3D;
     _graphics = this.CreateGraphics();
     _bottomLeft = new Point(0, height);
     _bottomRight = new Point(width, height);
     _tankCapacity = tankCapacity;
     _currentWaterLevel = EMPTY;
     _itsPump = new Pump(this, pumpCapacity);
     _penWidth = this.Height/(_tankCapacity/_itsPump.PumpingCapacity);
     if(_penWidth < 1) _penWidth = 1;
     _whitePen = new Pen(Color.White, _penWidth);
     _bluePen = new Pen(Color.Blue, _penWidth);
     _highLevelSensor = new WaterLevelSensor(tankCapacity - pumpCapacity, EMPTY);
     _highLevelSensor.SensorMode = WaterLevelSensor.Mode.HighLevelIndicator;
     _highLevelSensor.Fill += new WaterLevelEventHandler(_itsPump.FillTankEventHandler);
     _highLevelSensor.Full += new WaterLevelEventHandler(_itsPump.FullTankEventHandler);
     _lowLevelSensor = new WaterLevelSensor(pumpCapacity, EMPTY);
     _lowLevelSensor.SensorMode = WaterLevelSensor.Mode.LowLevelIndicator;
     _lowLevelSensor.Drain += new WaterLevelEventHandler(_itsPump.DrainTankEventHandler);
     _lowLevelSensor.Empty += new WaterLevelEventHandler(_itsPump.EmptyTankEventHandler);
  }

  public void ChangeWaterLevel(int amount){
     _lowLevelSensor.WaterLevelChange(amount);
     _highLevelSensor.WaterLevelChange(amount);
     _currentWaterLevel += amount;
     _lastWaterLevel = _currentWaterLevel;
     this.ChangeVisualLevel(amount);
  }

  private void ChangeVisualLevel(int amount){
     if(amount > 0){
      _graphics.DrawLine(_bluePen, _bottomLeft, _bottomRight);
      _bottomLeft.Y -= _penWidth;
      _bottomRight.Y -= _penWidth;

     } else{
      _graphics.DrawLine(_whitePen, _bottomLeft, _bottomRight);
      _bottomLeft.Y += _penWidth;
      _bottomRight.Y += _penWidth;
      Delay(30000000);
     }

  } // end ChangeVisualLevel method

  private void Delay(long ticks){
     for(long i = 0; i<ticks; i++){
      ;
     }
  }
} // end class definition
```

Referring to example 15.10 — the WaterTank class is an aggregate of a Pump and two WaterLevelSensors. It also provides a visual representation of a water tank by animating the rising and falling water level via blue and white lines drawn on a Panel. Most of the action occurs in three methods: InitializeComponents(), ChangeWaterLevel(), and ChangeVisualLevel(). (**Note:** An attempt is made to keep the visual filling animation in step with the tank's water level, however, when the value of _penWidth reaches 1, the animation gets a little goofy!)

The WaterLevelSensor objects are created in the InitializeComponents() method. One is designated as the _highLevelSensor and the other the _lowLevelSensor. Each sensor's SetPoint is set via its constructor followed by its SensorMode property. Next, the Pump's event handler methods are registered with each sensor's respective events.

Water is added to the tank via the ChangeWaterLevel() method. This in turn makes a call to each sensor's WaterLevelChange() method. The tank's level values are adjusted and finally its visual state is changed with a call to its ChangeVisualLevel() method. The Delay() method is used to slow down the draining animation so you can watch the water level drop.

15.11 WaterSystemApp.cs


```
using System;
using System.Windows.Forms;
using System.Drawing;

public class WaterSystemApp : Form {

 private FlowLayoutPanel _panel;
 private Button _button;
 private WaterTank _tank;

 public WaterSystemApp(){
    this.InitializeComponents();
 }

 public void InitializeComponents(){
    _tank = new WaterTank();
    _button = new Button();
    _button.Text = "Add Water";
    _button.Click += new EventHandler(this.AddWaterButtonClick);
    _button.Dock = DockStyle.Bottom;
    _panel = new FlowLayoutPanel();
    _panel.SuspendLayout();
    _panel.FlowDirection = FlowDirection.TopDown;
    _panel.AutoSize = true;
    _panel.AutoSizeMode = AutoSizeMode.GrowAndShrink;
    _panel.Height = _tank.Height + _button.Height + 75;
    _panel.Controls.Add(_tank);
    _panel.Controls.Add(_button);
    this.SuspendLayout();
    this.Text = "Water System";
    this.Height = _panel.Height;
    this.Width = _tank.Width;
    this.Controls.Add(_panel);
    _panel.ResumeLayout();
    this.ResumeLayout();
 }

 public void AddWaterButtonClick(object sender, EventArgs e){
     _tank.ChangeWaterLevel(_tank.FillRate);
 }

 public static void Main(){
    Application.Run(new WaterSystemApp());
 }
} // end WaterSystemApp class definition
```

Referring to example 15.11 — the WaterSystemApp class extends Form and provides the user interface for the water system application. It creates a FlowLayoutPanel and adds to it the WaterTank, which is itself a panel, and a button. Each time the button is clicked, water is added to the tank in an amount equal to the tank's *FillRate* property. The WaterTank.FillRate property is read-only and equals the value of its pump's PumpingCapacity. Figure 15-6 shows the results of running this program. However, you'll learn more from the program by running it and seeing for yourself how the events actually work. Experimenting with different tank dimensions and pumping capacities is left as an exercise.

Naming Conventions

If you'll pause for a moment to consider the previous two custom event examples, you'll notice a few similarities in the names given to certain components and methods. It helps to clarify the purpose of each component or method by adopting the following or similar naming convention.

- Add the suffix "EventArgs" to your event argument class names.
- Add the suffix "EventHandler" to your event-handler delegate names.
- Add the prefix "On" to the event name for the method that fires the event. *(i.e.*, OnFill())
- Add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods. (*i.e.*, FillTankEventHandler() or AddWaterButtonClick())

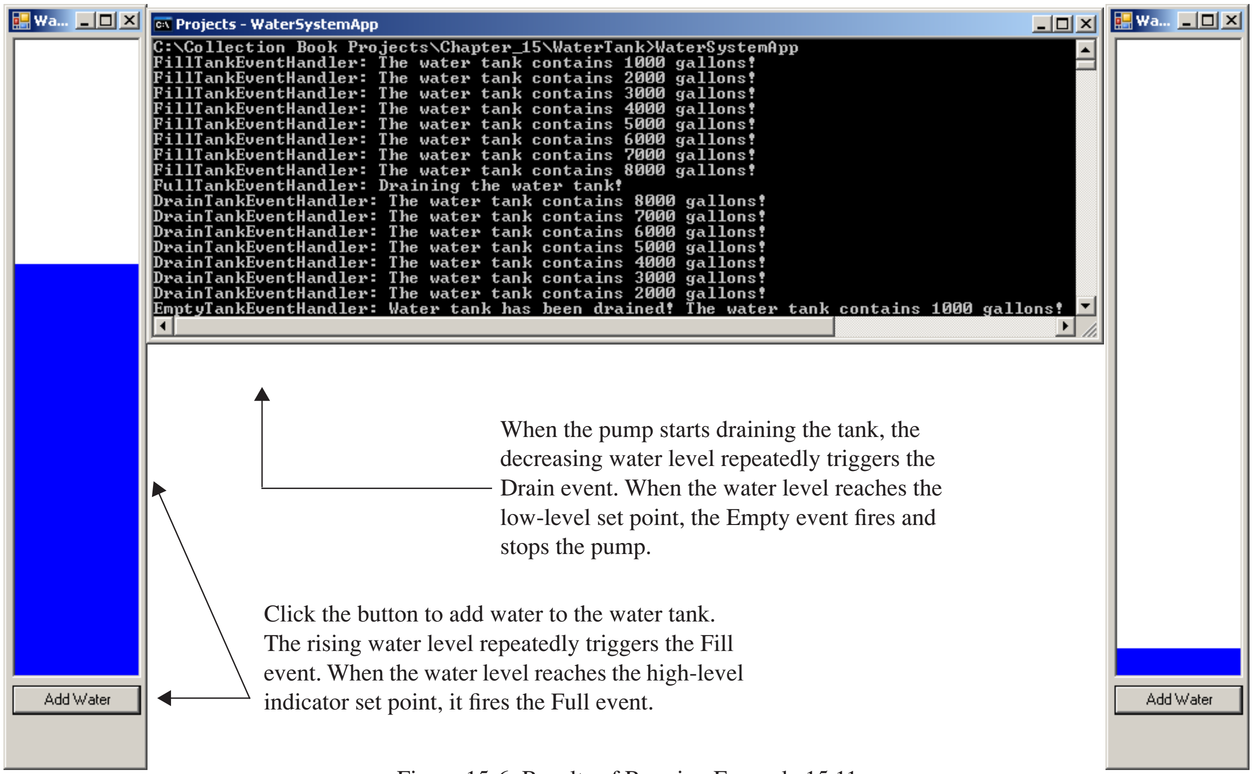


Figure 15-6: Results of Running Example 15.11

Final Thoughts On Extending The EventArgs Class

In the previous two programming examples, I created custom event argument classes by extending the EventArgs class, but this was not strictly necessary, since I didn't use any of the functionality provided by the EventArgs class. In fact, the EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

Summary

You need two logical components to implement the event processing model: 1) an event producer (*publisher*), and 2) an event consumer (*subscriber*). A *delegate* type specifies the authorized method signature for event handler methods. A delegate object maintains a list of event subscribers in the form of references to event handler methods. An event's subscriber list is initially empty until the first subscriber has been added to it with the '+=' operator. Event handler methods must conform to the method signature specified by an event's delegate type.

It helps to clarify the purpose of each component or method if you adopt the following or similar naming convention: add the suffix "EventArgs" to your event argument class name, add the suffix "EventHandler" to your event handler delegate names, add the prefix "On" to the event name for the method that fires the event, and finally, add the suffix "Handler" or optionally "ClassName + EventName" to your event handler methods.

It's not necessary to extend the System.EventArgs class to create custom event argument classes. The EventArgs class does nothing except provide a future evolutionary path for the .NET event API by serving as the base class for all the .NET event argument classes.

References

Microsoft Developer Network (MSDN) *.NET Framework 3.0, 3.5, and 4.0 Reference Documentation* [www.msdn.com]

Rick Miller. C# For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming. Pulp Free Press, 2008, ISBN-10: 1-932504-07-9 ISBN-13: 9781932504071

Notes

Chapter 16

Yashica Mat 124G

Old Bikes & Surfboards

Collections And Events

Learning Objectives

- Use the ObservableCollection<T> class in a program
- Use the BindlingList<T> class in a program
- Extract event information from a NotifyCollectionChangedEventArgs object
- Create event handlers that respond to the CollectionChanged Event
- Respond to events generated by a BindingList<T> object
- Implement the INotifyPropertyChanged interface on a user-defined type
- Bind a BindingList<T> object to a ListBox and a DataGridView
- Subclass the BindingList<T> class to gain access to its protected members
- Extract event information from an AddingNewEventArgs object
- Extract event information from a ListChangedEventArgs object

Introduction

By combining collections and events you get a powerful combination that lets your code respond automatically to collection state changes. There are, generally speaking, two ways of working with event-enabled collections: 1) use existing collection classes found in the .NET framework, and 2) create your own custom collection classes that publish unique events. In this chapter I am going to focus on several pre-existing event-enabled collection classes found in the .NET collections framework: the ObservableCollection<T> class and the BindingList<T> class.

The ObservableCollection<T> class allows you to respond to collection state changes via an event handler that you assign to its CollectionChanged event. I'll demonstrate the use of the ObservableCollection<T> class in a simple console application.

The BindingList<T> class can be used stand-alone or as a base class to create a two-way databinding mechanism. In this chapter I will demonstrate one-way databinding between a BindingList<T> subclass called SortableBindingList<T> and a ListBox. Next, I'll show you how to implement two-way databinding between a DataGridView control and a SortableBindingList<T> object. To completely understand the BindingList<T> examples you'll need to be familiar with Windows forms programming. If you're new to Windows forms programming please refer to my book *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming*, chapter 12. (See References section for full citation.)

ObservableCollection<T>

The System.Collections.ObjectModel.ObservableCollection<T> class allows you to respond automatically to collection state changes of which there are two types: 1) a change to the contents of the collection itself which occurs when adding or removing items, and 2) a change to one of the collection's properties of which there are three: Count, Item, and Items. Figure 16-1 gives the UML class diagram for the ObservableCollection<T> class.

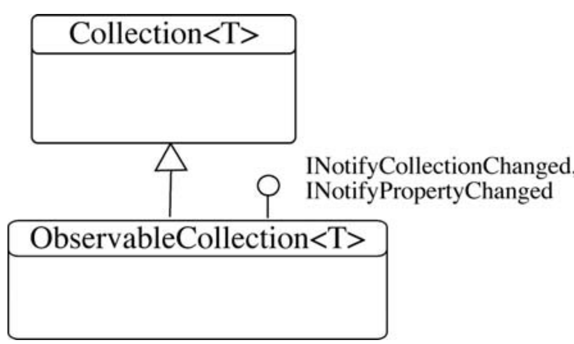


Figure 16-1: ObservableCollection<T> Class Diagram

Referring to figure 16-1 — the ObservableCollection<T> class extends the Collection<T> class and implements the INotifyCollectionChanged and INotifyPropertyChanged interfaces. The following sections describe in more detail the functionality provided by the Collection<T> class and the two interfaces.

Functionality Provided by the Collection<T> Class

The System.Collections.ObjectModel.Collection<T> class serves as the base class for the ObservableCollection<T> class and provides the bulk of its functionality. The Collection<T> class implements the IList, IList<T>, ICollection, ICollection<T>, IEnumerable, and IEnumerable<T> interfaces. This means, among other things, that an object of type ObservableCollection<T> can be accessed like an array and can be enumerated with the `foreach` statement.

Functionality Provided by the INotifyCollectionChanged Interface

The System.Collections.Specialized.INotifyCollectionChanged interface declares the CollectionChanged event which fires when the contents of a collection changes. Handle CollectionChanged events with a NotifyCollectionChangedEventHander delegate method which has the following method signature:

```
public void HandlerMethodName(object sender, NotifyCollectionChangedEventArgs e){
   // event handler code goes here
}
```

The NotifyCollectionChangedEventArgs object contains a wealth of information related to the CollectionChanged event in the form of properties which I've summarized in table 16.1 below.

Property	Description
Action	The action that caused the event. The type is a NotifyCollectionChangedAction enumeration which has the following values: Add, Remove, Replace, Move, and Reset.
NewItems	A list of the new items associated with the change.
NewStartingIndex	The starting index where the change occurred.
OldItems	A list of the old items associated with the change.
OldStartingIndex	The starting index where a Move, Remove, or Replace action took place.

Table 16-1: NotifyCollectionChangedEventArgs Properties

Functionality Provided by the INotifyPropertyChanged Interface

The System.ComponentModel.INotifyPropertyChanged interface declares the PropertyChanged event which fires when one or more of an object's properties change value. You can respond to PropertyChanged events with a PropertyChangedEventHandler delegate method which has the following method signature:

```
public void HandlerMethodName(object sender, PropertyChangedEventArgs e){
   // event handler code goes here
}
```

The PropertyChangedEventArgs declares the PropertyName property to indicate the name of the property that changed. A PropertyName value of null or String.Empty indicates all an object's properties changed.

ObservableCollection<T> Example Program

This section offers a short demonstration of the ObservableCollection<T> class. Three source files comprise the example: Person.cs and PersonKey.cs which I borrowed from chapter 11 remains unchanged for this example, and ObservableCollectionDemo.cs. Example 16.1 lists the Person class.

16.1 Person.cs

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
```



```
   private DateTime _birthday;
   private Guid _dna;

   public Person(){}

   public Person(String firstName, String middleName, String lastName,
                 Sex gender, DateTime birthday, Guid dna){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = dna;
   }

   public Person(String firstName, String middleName, String lastName,
                 Sex gender, DateTime birthday){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = Guid.NewGuid();
   }

   public Person(Person p){
      FirstName = p.FirstName;
      MiddleName = p.MiddleName;
      LastName = p.LastName;
      Gender = p.Gender;
      Birthday = p.Birthday;
      DNA = p.DNA;
   }

   // public properties
   public String FirstName {
     get { return _firstName; }
     set { _firstName = value; }
   }

   public String MiddleName {
     get { return _middleName; }
     set { _middleName = value; }
   }

   public String LastName {
     get { return _lastName; }
     set { _lastName = value; }
   }

   public Sex Gender {
     get { return _gender; }
     set { _gender = value; }
   }

   public DateTime Birthday {
     get { return _birthday; }
     set { _birthday = value; }
   }

   public Guid DNA {
     get { return _dna; }
     set { _dna = value; }
   }

   public int Age {
    get {
       int years = DateTime.Now.Year - _birthday.Year;
       int adjustment = 0;
       if(DateTime.Now.Month < _birthday.Month){
          adjustment = 1;
       }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
            adjustment = 1;
        }
     return years - adjustment;
     }
   }

   public String FullName {
     get { return FirstName + " " + MiddleName + " " + LastName; }
   }
```



```
  public String FullNameAndAge {
    get {  return FullName + " " + Age; }
  }

  protected String SortableName {
    get {  return LastName + FirstName + MiddleName; }
  }

  public PersonKey Key {
    get {  return new PersonKey(this.ToString()); }
  }

  public override String ToString(){
    return (FullName + "  " + Gender + "  " + Age + " " + DNA);
  }

  public override bool Equals(object o){
    if(o == null) return false;
    if(typeof(Person) != o.GetType()) return false;
    return this.ToString().Equals(o.ToString());
  }

  public override int GetHashCode(){
    return this.ToString().GetHashCode();
  }

  public static bool operator ==(Person lhs, Person rhs){
    return lhs.Equals(rhs);
  }

  public static bool operator !=(Person lhs, Person rhs){
    return !(lhs.Equals(rhs));
  }

  public int CompareTo(object obj){
    if((obj == null) || (typeof(Person) != obj.GetType()))  {
      throw new ArgumentException("Object is not a Person!");
    }
    return this.SortableName.CompareTo(((Person)obj).SortableName);
  }

  public int CompareTo(Person p){
    if(p == null){
      throw new ArgumentException("Cannot compare null objects!");
    }
    return this.SortableName.CompareTo(p.SortableName);
  }
} // end Person class
```

Referring to example 16.1 — the Person class remains unchanged from chapter 11. Example 16.2 lists the PersonKey class code.

16.2 PersonKey.cs

```
using System;

public class PersonKey : IEquatable<String>, IComparable, IComparable<PersonKey> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }
```



```

    public int CompareTo(object obj){
     return _keyString.CompareTo(obj);
    }


    public int CompareTo(PersonKey pk){
      return _keyString.CompareTo(pk._keyString);
    }
}
```

Referring to example 16.2 — the PersonKey class also remains unchanged from chapter 11. Example 16.3 lists the ObservableCollectionDemo application.

16.3 ObservableCollectionDemo.cs

```
1   using System;
2   using System.Collections.ObjectModel;
3   using System.ComponentModel;
4   using System.Collections.Specialized;
5
6   public class ObservableCollectionDemo {
7
8     private ObservableCollection<Person> _oc = null;
9
10    public ObservableCollectionDemo(){
11      _oc = new ObservableCollection<Person>();
12      _oc.CollectionChanged += CollectionChangedHandler;
13    }
14
15    public void InitializeCollection(){
16       _oc.Add(new Person("Rick", "Warren", "Miller", Person.Sex.MALE, new DateTime(1961, 2, 3),
17                          Guid.NewGuid()));
18       _oc.Add(new Person("Steve", "Jacob", "Hester", Person.Sex.MALE, new DateTime(1972, 1, 1),
19                          Guid.NewGuid()));
20       _oc.Add(new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE, new DateTime(1974, 8, 8),
21                          Guid.NewGuid()));
22       _oc.Add(new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE, new DateTime(1970, 5, 6),
23                          Guid.NewGuid()));
24       _oc.Add(new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE, new DateTime(1983, 2, 3),
25                          Guid.NewGuid()));
26       _oc.Add(new Person("Kyle", "Victor", "Miller", Person.Sex.MALE, new DateTime(1986, 10, 15),
27                          Guid.NewGuid()));
28    }
29
30    public void CollectionChangedHandler(object sender, NotifyCollectionChangedEventArgs e){
31
32       if(e.NewItems != null){
33         foreach(Person p in e.NewItems){
34           Console.Write("Collection changed. New Person Added: ");
35           Console.WriteLine(p.FullNameAndAge);
36         }
37        } else{
38           foreach(Person p in e.OldItems){
39             Console.Write("Collection changed. Person Removed: ");
40             Console.WriteLine(p.FullNameAndAge);
41           }
42        }
43    }
44
45    public void DeleteSomeItems(){
46      _oc.RemoveAt(0);
47      _oc.RemoveAt(1);
48    }
49
50    public static void Main(){
51      ObservableCollectionDemo pocd = new ObservableCollectionDemo();
52      pocd.InitializeCollection();
53      pocd.DeleteSomeItems();
54    }
55  }
```

Referring to example 16.3 — this program demonstrates how to respond to the CollectionChanged event and extract information from the NotifyCollectionChangedEventArgs class in response to the event. On line 8 an ObservableCollection<Person> reference named _oc is declared. The constructor method on line 10 initializes the _oc reference and adds the CollectionChangedHandler method to its CollectionChanged event. The InitializeCollection() method on line 15 simply adds six Person objects to the collection.

The CollectionChangedHandler() method definition begins on line 30. The `if` statement on line 32 checks to see if the NewItems property is null. If not, it iterates over the NewItems list and writes the FullNameAndAge of each Person object to the console. The `else` clause iterates over the OldItems list and writes the same information to the

console. The OldItems list will be populated if changes are made to the collection such as Removing or Deleting items.

The DeleteSomeItems() method on line 45 just removes two elements from the collection.

The Main() method on line 50 creates a new instance of the ObservableCollectionDemo class followed by calls to the InitializeCollection() and DeleteSomeItems() methods. Figure 16-2 shows the results of running this program

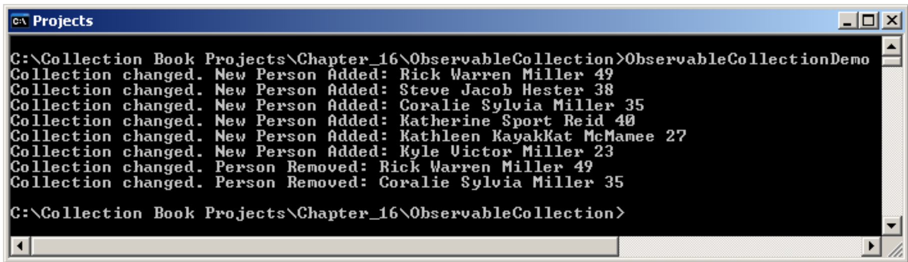


Figure 16-2: Results of Running Example 16.3

Referring to figure 16-2 — the CollectionChanged event fires each time a new Person object is added to the collection and when existing elements are deleted from the collection.

Quick Review

The ObservableCollection<T> class allows you to respond automatically to collection state changes by assigning event handlers to its CollectionChanged event. The NotifyCollectionChangedEventArgs object contains a wealth of information related to the CollectionChanged event in the form of properties which include the Action that occurred on the collection, the NewItems list, and the OldItems list.

BindingList<T>

The BindingList<T> collection is a complex creature that allows you to implement a databinding between the collection and a GUI component like a TextBox, ListBox, or DataGrid component.

Databindings can be one-way or two-way. In a one-way databinding scenario, a change to a databound collection will automatically reflect in the associated GUI component. For example, if a BindingList<T> object is used as the datasource for a ListBox, additions or deletions to the collection will be automatically reflected in the ListBox display. In a two-way databinding scenario, a change to the collection will be automatically reflected in the GUI component and a change to the GUI component will be automatically propagated to the collection.

Figure 16-3 shows the UML class diagram for the BindingList<T> class.

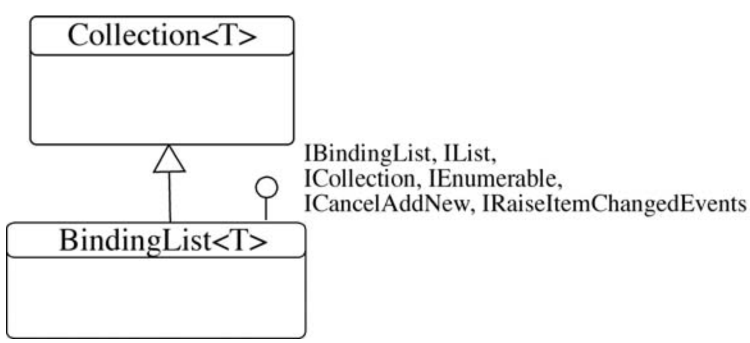


Figure 16-3: BindingList<T> Class Diagram

Referring to figure 16-3 — the BindingList<T> class extends the Collection<T> class and implements the IBindingList, IList, ICollection, IEnumerable, ICancelAddNew, and IRaiseItemChangedEvents interfaces. The following sections discus the functionality provided by these interfaces in greater detail.

Functionality Provided by the Collection<T> Class

The System.Collections.ObjectModel.Collection<T> class serves as the base class for the BindingList<T> class and provides the bulk of its functionality. The Collection<T> class implements the IList, IList<T>, ICollection, ICollection<T>, IEnumerable, and IEnumerable<T> interfaces. This means, among other things, that an object of type BindingList<T> can be accessed like an array and can be enumerated with the `foreach` statement.

Functionality Provided by the ICancelAddNew Interface

The System.ComponentModel.ICancelAddNew interface adds transactional capability to the BindingList<T> class so that the addition of new items to the list can be either committed or rolled back. The ICancelAddNew interface declares two methods: CancelNew() and EndNew(). The CancelNew() method will discard a pending item from the collection while the EndNew() method commits the new item to the collection.

Functionality Provided by the IRaiseItemChangedEvents Interface

The System.ComponentModel.IRaiseItemChangedEvents interface declares a singular property named RaisesItemChangedEvents which returns a bool value if the object that implements the interface raises ListChanged events when one or more of its properties change value.

One-Way Databinding Example

In this section I will present a one-way databinding example using the BindingList<T> class as the datasource and a GUI application that lists the items in the binding list in a ListBox. This example uses the Person and PersonKey classes listed in example 16.1 and 16.2. As you'll see shortly, the BindingList<Person> object will be assigned as the datasource for the ListBox control. Additions and deletions made to the BindingList<Person> collection will be reflected in the ListBox contents.

The BindingList<T> class could be used straight-up, but most of its members are protected, which means to access the majority of its functionality you really need to subclass the collection to create your own custom binding list. I had to subclass the collection to gain access to its protected Items property so I could sort the collection. Example 16.4 gives the code for a SortableBindingList<T> class.

16.4 SortableBindingList.cs

```
using System;
using System.ComponentModel;
using System.Collections.Generic;

public class SortableBindingList<T> : BindingList<T> {

  public void Sort(){
    ((List<T>)Items).Sort();
  }
}
```

Referring to example 16.4 — it doesn't look like much but there's a lot going on here. I'm creating a custom BindingList<T> class by extending BindingList<T>. I'm happy with the majority of the functionality the straight BindingList<T> class provides but because its Items property is protected I must extend the class to gain access to it so I can sort the list. The call to Items.Sort() sorts the lists according to the natural ordering provided by the objects in the list. The Items property must be cast to a List<T> for the Sort() method to be recognized. The public Sort() method allows me to sort the collection from a client program.

Example 16.5 lists the code for the BindingListDemo program. This is a Windows Forms program that displays a window with a ListBox control along with several Labels, TextBoxes, and Buttons. The program allows you to create, edit, and delete Person objects and display the items of a BindingList<Person> collection in a ListBox control.

16.5 BindingListDemo.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;

public class BindingListDemo : Form {

#region Fields
   private SortableBindingList<Person> _personList;
   private TableLayoutPanel _mainPanel;
   private TableLayoutPanel _subPanel1;
   private TableLayoutPanel _subPanel2;
   private ListBox _listBox;
   private Label _label1;
   private Label _label2;
   private Label _label3;
   private Label _label4;
   private Label _label5;
   private TextBox _textbox1;
   private TextBox _textbox2;
   private TextBox _textbox3;
   private DateTimePicker _dateTimePicker;
   private RadioButton _radioButton1;
   private RadioButton _radioButton2;
   private GroupBox _groupBox;
   private Button _button1;
   private Button _button2;
   private Button _button3;
   private Button _button4;
   private Button _button5;
   private Button _button6;
 #endregion

 #region Constructor
   public BindingListDemo(){
     InitializeComponent();
     InitializeBindingList();
     SetupListBox();
   }
 #endregion

 #region InitializationMethods
   private void InitializeComponent(){
     _mainPanel = new TableLayoutPanel();
     _mainPanel.RowCount = 2;
     _mainPanel.ColumnCount = 2;
     _subPanel1 = new TableLayoutPanel();
     _subPanel1.RowCount = 5;
     _subPanel1.ColumnCount = 2;
     _subPanel2 = new TableLayoutPanel();
     _subPanel2.RowCount = 2;
     _subPanel2.ColumnCount = 3;

     _listBox = new ListBox();
     _listBox.Height = 200;
     _listBox.Width = 200;

     _label1 = new Label();
     _label1.TextAlign = ContentAlignment.MiddleRight;
     _label1.Text = "First Name:";
     _label2 = new Label();
     _label2.TextAlign = ContentAlignment.MiddleRight;
     _label2.Text = "Middle Name:";
     _label3 = new Label();
     _label3.TextAlign = ContentAlignment.MiddleRight;
     _label3.Text = "Last Name:";
     _label4 = new Label();
     _label4.TextAlign = ContentAlignment.MiddleRight;
     _label4.Text = "Birthday:";
     _label5 = new Label();
     _label5.TextAlign = ContentAlignment.MiddleRight;
     _label5.Text = "Sex:";

     _textbox1 = new TextBox();
     _textbox2 = new TextBox();
     _textbox3 = new TextBox();

     _dateTimePicker = new DateTimePicker();
```



```

      _radioButton1 = new RadioButton();
      _radioButton1.Text = "Male";
      _radioButton1.Checked = true;
      _radioButton1.Location = new Point(10, 10);

      _radioButton2 = new RadioButton();
      _radioButton2.Text = "Female";
      _radioButton2.Location = new Point(10, 30);

      _groupBox = new GroupBox();
      _groupBox.Controls.Add(_radioButton1);
      _groupBox.Controls.Add(_radioButton2);
      _groupBox.Height = 75;
      _groupBox.Width = 150;

      _button1 = new Button();
      _button1.Text = "Clear";
      _button1.Click += ClearButton_Handler;

      _button2 = new Button();
      _button2.Text = "Submit";
      _button2.Click += SubmitButton_Handler;

      _button3 = new Button();
      _button3.Text = "Next";
      _button3.Click += NextButton_Handler;

      _button4 = new Button();
      _button4.Text = "Delete";
      _button4.Click += DeleteButton_Handler;

      _button5 = new Button();
      _button5.Text = "Edit";
      _button5.Click += EditButton_Handler;

      _button6 = new Button();
      _button6.Text = "Sort";
      _button6.Click += SortButton_Handler;

      _subPanel1.Controls.Add(_label1);
      _subPanel1.Controls.Add(_textbox1);
      _subPanel1.Controls.Add(_label2);
      _subPanel1.Controls.Add(_textbox2);
      _subPanel1.Controls.Add(_label3);
      _subPanel1.Controls.Add(_textbox3);
      _subPanel1.Controls.Add(_label4);
      _subPanel1.Controls.Add(_dateTimePicker);
      _subPanel1.Controls.Add(_label5);
      _subPanel1.Controls.Add(_groupBox);

      _subPanel2.Controls.Add(_button1);
      _subPanel2.Controls.Add(_button3);
      _subPanel2.Controls.Add(_button4);
      _subPanel2.Controls.Add(_button5);
      _subPanel2.Controls.Add(_button6);
      _subPanel2.Controls.Add(_button2);

      _mainPanel.Controls.Add(_listBox);
      _mainPanel.Controls.Add(_subPanel1);
      _mainPanel.Controls.Add(_subPanel2);
      _mainPanel.SetCellPosition(_subPanel2, new TableLayoutPanelCellPosition(1, 2));

      _subPanel1.Dock = DockStyle.Fill;
      _subPanel2.Dock = DockStyle.Fill;
      _subPanel2.Anchor = AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Top | AnchorStyles.Bottom;

      _mainPanel.Dock = DockStyle.Fill;

      this.Controls.Add(_mainPanel);
      this.Height = 300;
      this.Width = 550;
      this.MinimumSize = new Size(550, 300);
      this.MaximumSize = new Size(550, 300);
      this.Text = "Binding List Demo";
    }


    private void InitializeBindingList(){
       _personList = new SortableBindingList<Person>();
       _personList.AddingNew += AddingNew_Handler;
```

```
      _personList.ListChanged += ListChanged_Handler;
      _personList.AllowNew = true;
      _personList.AllowEdit = true;
      _personList.AllowRemove = true;
      _personList.RaiseListChangedEvents = true;
    }

    private void SetupListBox(){
      _listBox.DataSource = _personList;
      _listBox.DisplayMember = "FullNameAndAge";
      _listBox.SelectedIndexChanged += SelectedIndexChanged_Handler;
      _listBox.SelectionMode = SelectionMode.One;
    }

    #endregion

    #region UtilityMethods
    private void ClearEntryControls(){
      _textbox1.Text = string.Empty;
      _textbox2.Text = string.Empty;
      _textbox3.Text = string.Empty;
      _dateTimePicker.Value = DateTime.Now;
      _radioButton1.Checked = true;
    }

    private void UpdateEntryControls(int selectedIndex){
      _textbox1.Text = _personList[selectedIndex].FirstName;
      _textbox2.Text = _personList[selectedIndex].MiddleName;
      _textbox3.Text = _personList[selectedIndex].LastName;
      _dateTimePicker.Value = _personList[selectedIndex].Birthday;
      ConvertGenderToRadioButtonSelection(_personList[selectedIndex].Gender);
    }

    private Person.Sex ConvertRadioButtonToGender(){
      return _radioButton1.Checked? Person.Sex.MALE:Person.Sex.FEMALE;
    }

    private void ConvertGenderToRadioButtonSelection(Person.Sex gender){
      switch(gender){
        case Person.Sex.MALE: _radioButton1.Checked = true;
                   break;
        case Person.Sex.FEMALE: _radioButton2.Checked = true;
                   break;
      }
    }

    #endregion


    #region EventHandlerMethods
    public void SelectedIndexChanged_Handler(object sender, EventArgs e){
      UpdateEntryControls(_listBox.SelectedIndex);
    }

    public void ClearButton_Handler(object sender, EventArgs e){
      ClearEntryControls();
    }



    public void SubmitButton_Handler(object sender, EventArgs e){
      Person p = _personList.AddNew();
      if((p.FirstName == string.Empty) || (p.LastName == string.Empty)){
        MessageBox.Show("First Name and Last Name cannot be blank!");
        _personList.CancelNew(_personList.IndexOf(p));
      } else {
         ClearEntryControls();
        }
    }

    public void NextButton_Handler(object sender, EventArgs e){
      if(_personList.Count > 0){
      try{
        ++_listBox.SelectedIndex;
      }catch(ArgumentOutOfRangeException){
        //We tried to go beyond the bounds of the listbox index
        //reset SelectedIndex to zero.
        _listBox.SelectedIndex = 0;
      }
        UpdateEntryControls(_listBox.SelectedIndex);
      }else{
```



```
        MessageBox.Show("There are no items in the list!", "No Items Alert",
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
     }
   }

   public void DeleteButton_Handler(object sender, EventArgs e){
     if(_listBox.SelectedIndex > -1){
        DialogResult result = MessageBox.Show("Are you sure you want to delete this person?",
                                              "Delete Warning", MessageBoxButtons.OK,
                                              MessageBoxIcon.Exclamation);
        if(result == DialogResult.OK){
          _personList.RemoveAt(_listBox.SelectedIndex);
        }
        ClearEntryControls();
     }else{
        MessageBox.Show("There are no items to delete!", "No Items Alert",
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
     }
   }


   public void AddingNew_Handler(object sender, AddingNewEventArgs e){
      e.NewObject = new Person(_textbox1.Text, _textbox2.Text, _textbox3.Text,
                               ConvertRadioButtonToGender(), _dateTimePicker.Value);
   }


   public void ListChanged_Handler(object sender, ListChangedEventArgs e){
     switch(e.ListChangedType){
       case ListChangedType.ItemDeleted:
              MessageBox.Show("Item successfully deleted.");
              break;
       case ListChangedType.ItemChanged:
              MessageBox.Show("Item successfully updated.");
              break;
     }
   }


   public void EditButton_Handler(object sender, EventArgs e){
     if(_listBox.SelectedIndex > -1){

       //update the person's properties
      _personList[_listBox.SelectedIndex].FirstName = _textbox1.Text;
      _personList[_listBox.SelectedIndex].MiddleName = _textbox2.Text;
      _personList[_listBox.SelectedIndex].LastName = _textbox3.Text;
      _personList[_listBox.SelectedIndex].Birthday = _dateTimePicker.Value;
      _personList[_listBox.SelectedIndex].Gender = ConvertRadioButtonToGender();
      //then refresh the listbox to display our updated person
      ((CurrencyManager)_listBox.BindingContext[_personList]).Refresh();

     } else {
        MessageBox.Show("There are no items to edit!", "No Items Alert",
                        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
     }
   }


   public void SortButton_Handler(object sender, EventArgs e){
      _personList.Sort();
      ((CurrencyManager)_listBox.BindingContext[_personList]).Refresh();
      if(_personList.Count > 0){
        UpdateEntryControls(_listBox.SelectedIndex = 0);
      }
   }

  #endregion



 #region MainMethod

   [STAThread]
   public static void Main(){
     Application.Run(new BindingListDemo());
   }
 #endregion
}
```

Referring to example 16.5 — there's a lot of code here but most of the code is devoted to the GUI and is fairly straightforward to follow. I've organized the program into regions using the #region/#endregion directives. The first

region is the fields region where I declare a SortableBindingList<Person> reference named _personList. The remaining fields are Windows.Forms controls including TableLayoutPanels, a ListBox, assorted Labels, TextBoxes, and Buttons, RadioButtons and a GroupBox, and a DatePicker. The constructor method which begins on line 36 calls three methods: InitializeComponent(), InitializeBindingList(), and SetupListBox().

The InitializeComponent() method creates the GUI components and builds the application window, organizing the TableLayoutPanels and other controls. Most of the business logic performed by the application exists within the various event handler methods assigned to each button. These include the ClearButton_Handler(), SubmitButton_Handler(), NextButton_Handler(), DeleteButton_Handler(), EditButton_Handler(), and SortButton_Handler() methods which are grouped together in the EventHandlerMethods region. I'll discuss each of these methods in turn.

The InitializeBindingList() method which starts on line 158 creates an instance of SortableBindingList<Person> and assigns event handler methods to its AddingNew and ListChanged events. It then sets several list properties to `true` including AllowNew, AllowEdit, AllowRemove and RaiseListChangedEvents.

The SetupListBox() method assigns the _personList to the _listBox.DataSource property. It then specifies on line 170 to use the Person.FullNameAndAge property as the value to display in the list. If you don't specify a DisplayMember property it defaults to the object's ToString() representation. (And if you haven't overridden Object.ToString() your listbox will be quite boring.) Following this I assign an event handler method to respond to the _listBox.SelectedIndexChanged event. This occurs when you click on a different row from the one currently highlighted.

OK, next, there are several utility methods. These include the ClearEntryControls() method which clears the textboxes and sets the DatePicker and radio buttons to default values to make way for a new entry; the UpdateEntryControls() method which sets the value of the textboxes, DatePicker, and radio buttons to values corresponding to a person object in the datasource; the ConvertRadioButtonToGender() method which examines the value of the radio buttons and returns the corresponding Person.Sex enumeration value; and finally the ConvertGenderToRadioButtonSelection() method which sets the radio buttons based on the value of the Person.Sex enumeration argument.

The Main() method simply makes a call to the static Application.Run() method passing in an instance of the BindingListDemo class. Before discussing the operation of this program in more detail let's see how the program looks when running. Figure 16-4 shows how the program looks on startup.

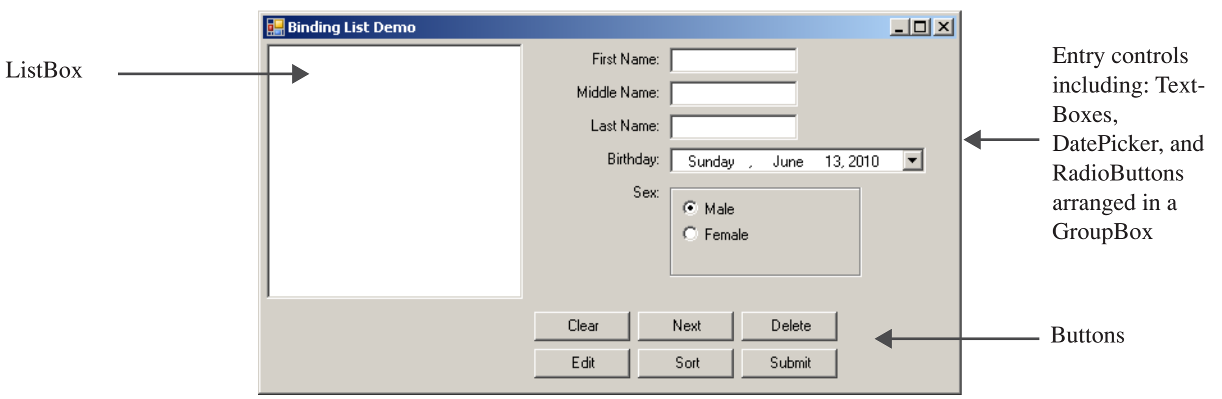


Figure 16-4: Binding List Demo Program on Startup

Referring to figure 16-4 — since the _personList, which is used as a datasource for the _listBox, is initially empty, the _listBox has nothing to display. To add a Person object to the _personList make the appropriate entry in the TextBoxes, set the DatePicker, and click the appropriate RadioButton, then click the Submit button. The Submit button's Click event will fire, which will send a notification to its assigned event handler method. Let's take a closer look at the SubmitButton_Handler() method that starts on line 221. On line 222 a new Person object is created with a call to the _personList.AddNew() method. The call to AddNew() fires the _personList.AddingNew event, which is handled by the AddingNew_Handler() method which begins on line 263. The AddingNew event signals the pending insertion of a new item into the BindingList. At this point in the AddingNew_Handler() method on line 264 I'm creating a new Person object using the values in the data entry fields and assigning it to the NewObject property of the AddingNewEventArgs object. When the AddingNew_Handler() method returns, control returns to the SubmitButton_Handler() method line 223 where I check to ensure the FirstName and LastName properties are not set

to String.Empty. If they are, I display a warning message box then roll back the submit transaction with a call on line 225 to the _personList.CancelNew() method. Otherwise, if all's well, I let the submit transaction pass and clear the entry controls with a call to the ClearEntryControls() method. Figures 16-5 through 16-7 show several names being entered into the application.

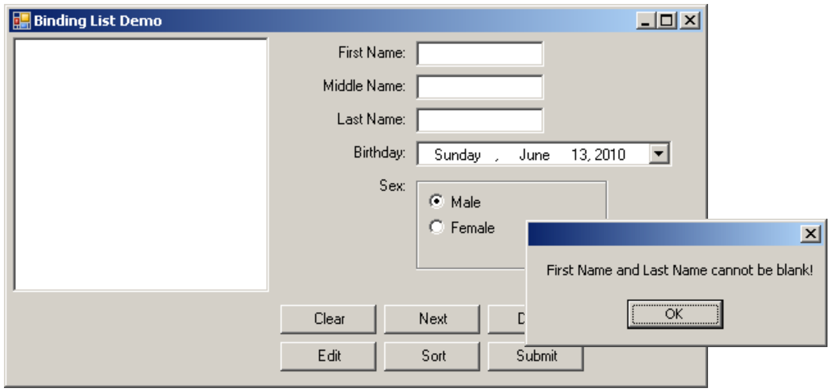


Figure 16-5: Warning MessageBox When First Name or Last Name TextBoxes are Empty on Submit

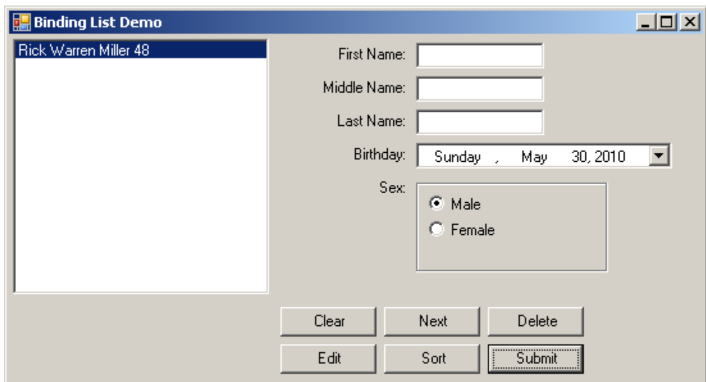


Figure 16-6: One Name Entered and Displayed in the ListBox

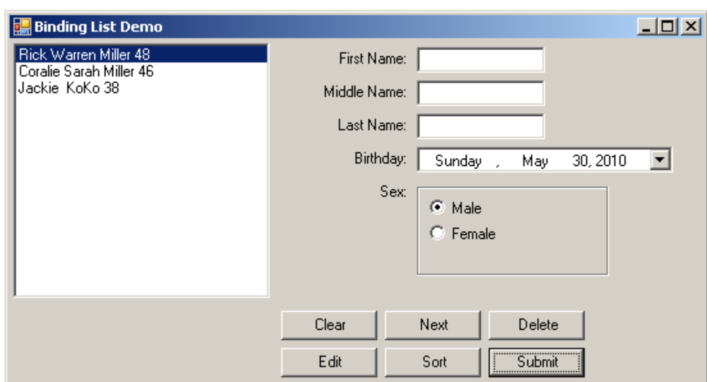


Figure 16-7: Three Names Entered and Displayed in the ListBox

Referring to figure 16-5 — if you click the Submit button with empty First Name and Last Name textboxes you'll receive a warning message box saying the fields cannot be blank. Once several names have been entered they can be sorted or deleted. Clicking the sort button fires the SortButton_Handler() method on line 300 which in turn calls _personList.Sort() followed by this line of code:

```
((CurrencyManager)_listBox.BindingContext[_personList]).Refresh();
```

This line of code refreshes the _listBox contents.

Two-Way Databinding Example

Two-way databinding is nice because not only do changes in the underlying datasource cause changes to the UI control, you can also edit the underlying datasource directly via the databound UI control. In this section I'm going to demonstrate the use of a DataGridView control databound to a BindingList<T> object. I'll use the Person class again but this time I want the Person class to generate PropertyChanged events when its properties are changed. The modified person class is listed in example 16.6.

16.6 Person.cs (Implementing the INotifyPropertyChanged interface.)

```
using System;
using System.ComponentModel;

public class Person : IComparable, IComparable<Person>, INotifyPropertyChanged {

  //enumeration
  public enum Sex { MALE, FEMALE} ;


  //event
  public event PropertyChangedEventHandler PropertyChanged;


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;


  public Person(){
    _firstName = string.Empty;
    _middleName = string.Empty;
    _lastName = string.Empty;
    _gender = Person.Sex.MALE;
    _birthday = DateTime.Now;
    _dna = Guid.NewGuid();
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
    get {  return _firstName; }
    set {  _firstName = value;
           NotifyPropertyChanged("FirstName");
        }
  }
```



```
public String MiddleName {
  get {  return _middleName; }
  set {  _middleName = value;
        NotifyPropertyChanged("MiddleName");
      }
}

public String LastName {
  get {  return _lastName; }
  set {  _lastName = value;
        NotifyPropertyChanged("LastName");
      }
}

public Sex Gender {
  get {  return _gender; }
  set {  _gender = value;
       NotifyPropertyChanged("Gender");
      }
}

public DateTime Birthday {
  get {  return _birthday; }
  set {  _birthday = value;
       NotifyPropertyChanged("Birthday");
      }
}

public Guid DNA {
  get {  return _dna; }
  set {  _dna = value;
       NotifyPropertyChanged("DNA");
  }
}

public int Age {
   get {
     int years = DateTime.Now.Year - _birthday.Year;
     int adjustment = 0;
    if(DateTime.Now.Month < _birthday.Month){
       adjustment = 1;
     }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
           adjustment = 1;
         }
    return years - adjustment;
  }
}

public String FullName {
  get {  return FirstName + " " + MiddleName + " " + LastName; }
}

public String FullNameAndAge {
  get {  return FullName + " " + Age; }
}

protected String SortableName {
  get {  return LastName + FirstName + MiddleName; }
}

public PersonKey Key {
  get {  return new PersonKey(this.ToString()); }
}

public override String ToString(){
  return (FullName + "  " + Gender + "  " + Age + " " + DNA);
}

public override bool Equals(object o){
  if(o == null) return false;
  if(typeof(Person) != o.GetType()) return false;
  return this.ToString().Equals(o.ToString());
}

public override int GetHashCode(){
  return this.ToString().GetHashCode();
}

public static bool operator ==(Person lhs, Person rhs){
  return lhs.Equals(rhs);
}
```

```
151
152   public static bool operator !=(Person lhs, Person rhs){
153     return !(lhs.Equals(rhs));
154   }
155
156   public int CompareTo(object obj){
157     if((obj == null) || (typeof(Person) != obj.GetType()))  {
158       throw new ArgumentException("Object is not a Person!");
159     }
160     return this.SortableName.CompareTo(((Person)obj).SortableName);
161   }
162
163   public int CompareTo(Person p){
164     if(p == null){
165       throw new ArgumentException("Cannot compare null objects!");
166     }
167     return this.SortableName.CompareTo(p.SortableName);
168   }
169
170   private void NotifyPropertyChanged(string propertyName){
171     if(PropertyChanged != null){
172       PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
173     }
174   }
175 } // end Person class
```

Referring to example 16.6 —Two primary changes have been made to the Person class. 1) it now implements the INotifyPropertyChanged interface which requires the addition of two new class members: a) on line 11 I've added an event named PropertyChanged, and b) on line 170 I've implemented the NotifyPropertyChanged() method which takes one argument of type String indicating the name of the changed property. The NotifyPropertyChanged() method checks to ensure there is at least one event handler assigned to the PropertyChanged event and if so signals a notification on line 172 with a call to the PropertyChanged() delegate passing in a reference to the current object (`this`) and a new instance of the PropertyChangedEventArgs class passing in the name of the property that was just changed, and 2) I have added field initialization code to the default constructor. I did this mostly because I needed the Guid _dna field to be initialized to a valid Guid so it would display properly in the DataGridView's DNA column.

The PersonKey class and SortableBindingList<T> classes remain unchanged from the previous example so I won't repeat that code here.

Example 16.7 lists the BindingListDataGridDemo class. The first thing you'll notice about this example is that it's shorter than the ListBox example but somewhat more mysterious.

16.7 BindingListDataGridDemo.cs (Demonstrating two-way databinding.)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;
using System.Data;

public class BindingListDataGridDemo : Form {

 #region Fields
  SortableBindingList<Person> _personList;
  DataGridView _dataGridView;
  TableLayoutPanel _mainPanel;
  TableLayoutPanel _buttonPanel;
  Button _button1;
  Button _button2;
 #endregion

 #region Constructor
   public BindingListDataGridDemo(){
     InitializeBindingList();
     InitializeComponent();
   }
 #endregion

 #region InitializationMethods
   private void InitializeBindingList(){
      _personList = new SortableBindingList<Person>();
      _personList.AddingNew += AddingNew_Handler;
      _personList.ListChanged += ListChanged_Handler;
      _personList.AllowNew = true;
      _personList.AllowEdit = true;
      _personList.AllowRemove = true;
      _personList.RaiseListChangedEvents = true;
   }
```



```
   private void InitializeComponent(){
     _mainPanel = new TableLayoutPanel();
     _mainPanel.RowCount = 2;
     _mainPanel.ColumnCount = 1;
     _mainPanel.Dock = DockStyle.Fill;

     _buttonPanel = new TableLayoutPanel();
     _buttonPanel.RowCount = 1;
     _buttonPanel.ColumnCount = 2;
     _buttonPanel.Dock = DockStyle.Fill;

     InitializeDataGridView();

     _button1 = new Button();
     _button1.Text = "Sort";
     _button1.Click += SortButton_Handler;

     _button2 = new Button();
     _button2.Text = "Delete";
     _button2.Click += DeleteButton_Handler;

     _buttonPanel.Controls.Add(_button1);
     _buttonPanel.Controls.Add(_button2);

     _mainPanel.Controls.Add(_dataGridView);
     _mainPanel.Controls.Add(_buttonPanel);

     this.Controls.Add(_mainPanel);
     this.Width = 850;
     this.Height = 250;
     this.Text = "BindingListDataGridDemo";
   }


   private void InitializeDataGridView(){
     _dataGridView = new DataGridView();
     _dataGridView.Dock = DockStyle.Fill;
     DataGridViewComboBoxColumn genderColumn = new DataGridViewComboBoxColumn();
     genderColumn.DataSource = Enum.GetValues(typeof(Person.Sex));
     genderColumn.DataPropertyName = "Gender";
     genderColumn.HeaderText = "Gender";
     _dataGridView.Columns.Add(genderColumn);
     _dataGridView.DataSource = _personList;
     _dataGridView.EditMode = DataGridViewEditMode.EditOnEnter;
     _dataGridView.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
     _dataGridView.DataBindingComplete += DataBindingComplete_Handler;
   }
  #endregion


  #region EventHandlerMethods

   public void AddingNew_Handler(object sender, AddingNewEventArgs e){
      e.NewObject = new Person();
      Console.WriteLine("New Person object created!");
   }

   public void ListChanged_Handler(object sender, ListChangedEventArgs e){
     switch(e.ListChangedType){
       case ListChangedType.ItemDeleted:
               Console.WriteLine("Item successfully deleted.");
               foreach(Person p in _personList){
                 Console.WriteLine(p);
               }
               break;
       case ListChangedType.ItemChanged:
               ((CurrencyManager)_dataGridView.BindingContext[ _personList]).Refresh();
               Console.Write("Item successfully updated. Property: " + e.PropertyDescriptor.Name );
               Console.WriteLine(" - Value: " + e.PropertyDescriptor.GetValue(_personList[ e.NewIndex]));

               break;
     }
   }

  public void SortButton_Handler(object sender, EventArgs e){
    _personList.Sort();
    ((CurrencyManager)_dataGridView.BindingContext[ _personList]).Refresh();
  }
```

```
117
118
119    public void DeleteButton_Handler(object sender, EventArgs e){
120        if(_personList.Count > 0){
121          _personList.RemoveAt(_dataGridView.CurrentRow.Index);
122          Console.WriteLine("Person object deleted!");
123        }
124    }
125
126    public void DataBindingComplete_Handler(object sender, EventArgs e){
127      _dataGridView.Columns["FullNameAndAge"].Visible = false;
128      _dataGridView.Columns["FullName"].Visible = false;
129      _dataGridView.Columns["Key"].Visible = false;
130      _dataGridView.Columns["DNA"].ReadOnly = true;
131      _dataGridView.Columns["DNA"].ToolTipText = "Read Only!";
132      _dataGridView.Columns["Birthday"].ToolTipText = "Format: mm/dd/yyyy";
133
134      for(int i=0; i<_dataGridView.Columns.Count; i++){
135        _dataGridView.Columns[i].Width = 100;
136      }
137
138      _dataGridView.Columns["DNA"].Width = 225;
139      _dataGridView.Columns["Age"].Width = 50;
140      _dataGridView.Columns["FirstName"].DisplayIndex = 0;
141      _dataGridView.Columns["MiddleName"].DisplayIndex = 1;
142      _dataGridView.Columns["LastName"].DisplayIndex = 2;
143    }
144
145   #endregion
146
147
148   #region MainMethod
149     [STAThread]
150     public static void Main(){
151       Application.Run(new BindingListDataGridDemo());
152     }
153   #endregion
154 }
```

Referring to example 16.7 — the primary components of this example are a SortableBindingList<Person> object, a DataGridView component, two TableLayoutPanels, and two Buttons. These are all declared in the Fields region. There's one constructor which calls two initialization methods: InitializeBindingList() and InitializeComponent(). The InitializeComponent() method sets up the GUI and along the way calls the InitializeDataGridView() method. Before going further let's see how the application looks when it's launched. Figure 16-8 shows the program upon startup.

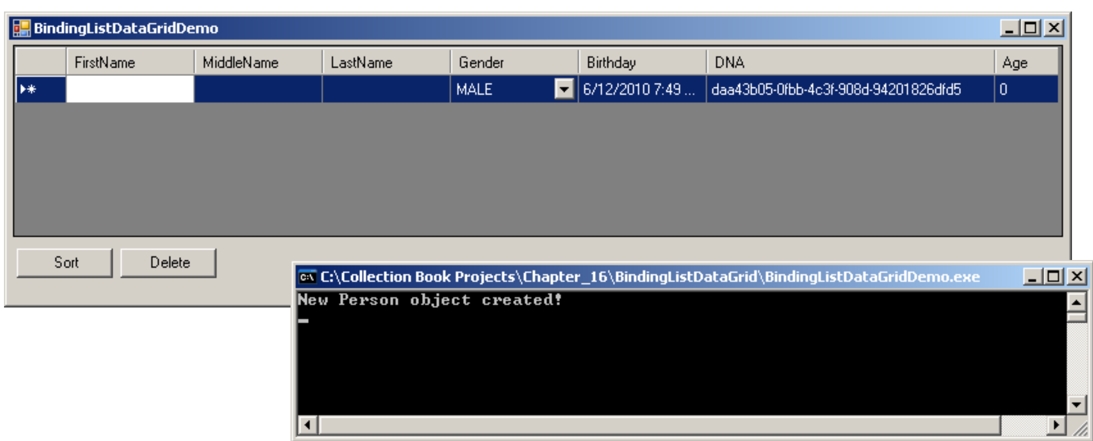


Figure 16-8: Binding List Data Grid Demo Application on Startup

Refer both to figure 16-8 and to the InitializeDataGridView() method on line 73 of example 16.7. First, some general comments about binding a datasource to a DataGridView component. The columns and column heading text will be inferred automatically by the public properties of the objects in the SortableBindingList<Person> object. However, you may not want or need to display all the possible columns because some of the properties of the Person class are either read-only or are meant for convenience purposes like the FullNameAndAge property. Also, you may need to create custom columns in the DataGridView to allow special handling of different data types. For example,

the Gender column shown above is a drop-down list or combo box. This is not the default way the Person.Gender property would be rendered as a column. It would normally be rendered as a textbox, or, more specifically, as a DataGridViewTextBoxColumn. You can still edit the Person.Gender property via the default textbox column but the combo box is the more natural choice of data entry when limited to a narrow choice of authorized values.

OK, the trick to customizing the columns shown in the DataGridView control is to create any custom columns you need, associate the corresponding properties, and then assign an event handler method to the DataGridView control's DataBindingComplete event. In this example I've assigned the DataBindingComplete_Handler() method which begins on line 126. Let's take a closer look at that method.

In order to remove columns or manipulate columns in the DataGridView control the databinding between the control and the datasource, in this case a SortableBindingList<Person> object, must be complete, otherwise the Columns property is not yet populated. Therefore, I postpone access to the _dataGridView.Columns property until the DataBindingComplete method fires. In the body of the DataBindingComplete_Handler() method I set the visibility of several columns I don't want to display to `false`. I set the "DNA" column ReadOnly property to `true`. I then set the ToolTipText property of several columns followed by an attempt to set the widths of all the columns to 100 in the body of the `for` loop. I then adjust the widths of the "DNA" and "Age" columns and then reorder the "FirstName", "MiddleName", and "LastName" columns by changing their DisplayIndex properties to the desired setting.

Alright, let's enter some data and follow the code. Figure 16-9 shows some data being entered and the underlying messages being written to the console.

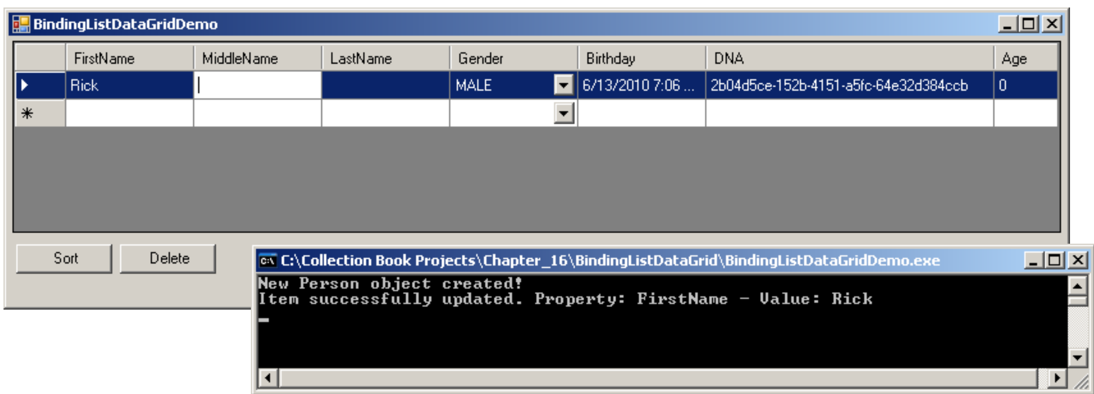


Figure 16-9: Data Being Entered into the DataGridView and the Resulting Messages Generated to the Console

Referring to figure 16-9 — when I enter the string "Rick" into the FirstName cell and either click or tab into the MiddleName cell, the string "Rick" is presented to the FirstName property of the Person object residing in the SortableBindingList at the row index associated with the first row, which in this case is row index 0 (i.e., _personList[0].FirstName). A change to the person object's FirstName property causes its PropertyChanged event to fire. This in turn results in a ListChanged event to fire on the _personList. If you now take a look at the ListChanged_Handler() method on line 96 you'll see that I'm interested specifically in when items are either deleted from the list (ItemDeleted) or when an item is changed (ItemChanged). I should point out now that when the program first starts up and the DataGridView component is first rendered and the first row is automatically added, this causes the _personList.AddingNew event to fire which is handled by the AddingNew_Handler() method. This creates a new Person object using the default constructor. Figure 16-10 shows the program state and resulting console output after adding another row to the DataGridView control. Note that to add another Person object to the list simply select the empty row underneath the one you're currently editing.

Let's examine the code behind the console message showing the name of the property that was just changed. The ItemChanged case begins on line 104. The first thing I do is refresh the _personList BindingContext with this line of code:

```
((CurrencyManager)_dataGridView.BindingContext[_personList]).Refresh();
```

I need to do this to get the Age column to update when a change is made to the Birthday column. Following the refresh, I write the name of the changed property to the console. It can be found in the ListChangedEventArgs.PropertyDescriptor.Name property. On the next line I write the value of the new property with the help of the PropertyDescriptor.GetValue() method which uses reflection to retrieve the just-changed property value from the Person object.

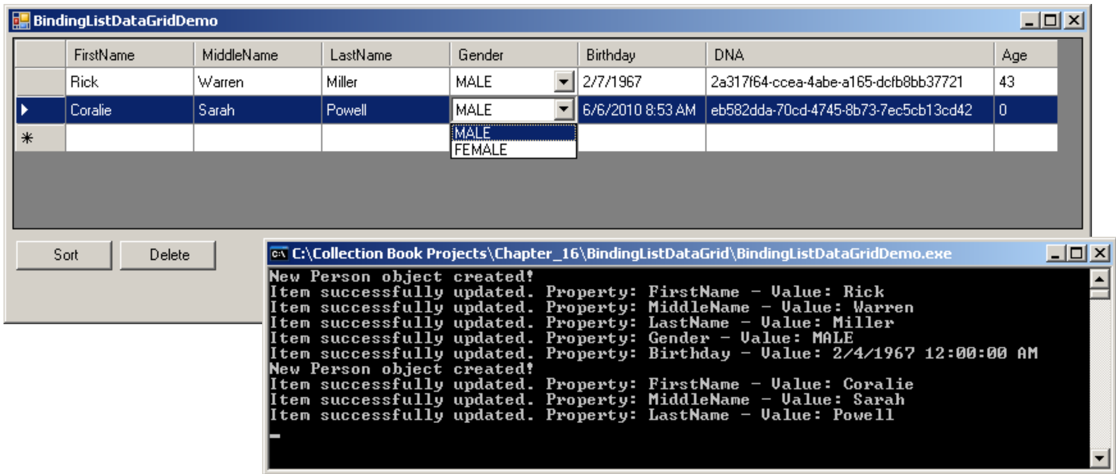


Figure 16-10: More Data Added to the DataGridView and Resulting Console Messages

Quick Review

The BindingList<T> class can be used standalone or subclassed to create a two-way databinding between itself and GUI components. The two events of interest on a BindingList<T> object are the AddingNew and ListChanged events. To gain full benefit from automatically fired events, the objects stored within the BindingList<T> collection must implement the INotifyProperty changed interface.

You'll need to subclass BindingList<T> to gain access to its protected members. You'll need to do this if you want to sort the list or perform other types of custom list manipulation that calls for access to its protected members.

You can respond to specific BindingList<T>.ListChanged events by inspecting the ListChangedEventArgs.ListChangedType property. Use the ListChangedEventArgs.PropertyDescriptor property to get specific information about a changed property and its value.

Summary

The ObservableCollection<T> class allows you to respond automatically to collection state changes by assigning event handlers to its CollectionChanged event. The NotifyCollectionChangedEventArgs object contains a wealth of information related to the CollectionChanged event in the form of properties which include the Action that occurred on the collection, the NewItems list, and the OldItems list.

The BindingList<T> class can be used standalone or subclassed to create a two-way databinding between itself and GUI components. The two events of interest on a BindingList<T> object are the AddingNew and ListChanged events. To gain full benefit from automatically fired events the objects stored within the BindingList<T> collection must implement the INotifyProperty changed interface.

You'll need to subclass BindingList<T> to gain access to its protected members. You'll need to do this if you want to sort the list or perform other types of custom list manipulation that calls for access to its protected members.

You can respond to specific BindingList<T>.ListChanged events by inspecting the ListChangedEventArgs.ListChangedType property. Use the ListChangedEventArgs.PropertyDescriptor property to get specific information about a changed property and its value.

To modify the automatically inferred column types and layout of a DataGridView component you must wait until the DataGridView.Columns property has been populated. This does not occur until the DataBindingComplete event has fired. Create and add custom columns to the DataGridView before databinding to a datasource. Then, change the layout of the columns and make other column property changes in your DataBindingComplete event handler method.

References

Microsoft Developer Network (MSDN) *.NET Framework 3.0, 3.5, and 4.0 Reference Documentation* [www.msdn.com]

Notes

Chapter 17

Contax T

Amsterdam Mounted Police

Collections And I/O

Learning Objectives

- *State the purpose of the [Serializable] attribute*
- *Use a BinaryFormatter to serialize a collection of objects*
- *Use a BinaryFormatter to deserialize a collection of objects*
- *Use an XMLFormatter to serialize a collection of objects to an XML file*
- *Use an XMLFormatter to deserialize a collection of objects*
- *Understand the limitations with using the XMLFormatter to serialize Dictionaries*
- *Write a custom XML Serializer*

Introduction

All but the most trivial software applications must preserve their data in some form or another. This chapter shows you how to preserve your application data to local files. These files might be located on a hard drive, a floppy disk, a USB drive, or some other type of media connected to your computer. In most cases, the type of media is of no concern to you because the operating system, and the storage device's driver software, handle the machine-specific details. All you need to know to conduct file Input/Output (I/O) operations is a handful of .NET Framework classes. The operating system does the rest.

You're going to learn a lot of cool things in this chapter, like how to manipulate files and directories, how to serialize and deserialize objects to disk, how to read and write text files, how to perform random access file I/O, how to write log files, and finally, how to use an OpenFileDialog to locate and open files. You will be surprised to learn you can do all these things with only a small handful of classes, structures, and enumerations, most of which are found in the System.IO namespace.

When you finish this chapter, you will have reached an important milestone in your C# programming career — you will be able to write applications that save data to disk. You will find this to be a critical skill to have in your programmer's toolbox.

Manipulating Directories And Files

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*. I say "in most cases" because it is entirely possible to write data to an absolute or random position on a device, depending of course on what type of storage medium you're talking about. (*i.e.,* A disk drive works differently than a tape drive.)

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data. The file management services provided by the operating system are part of a set of layered services that make it possible to build complex computing systems, as Figure 17-1 partially illustrates.

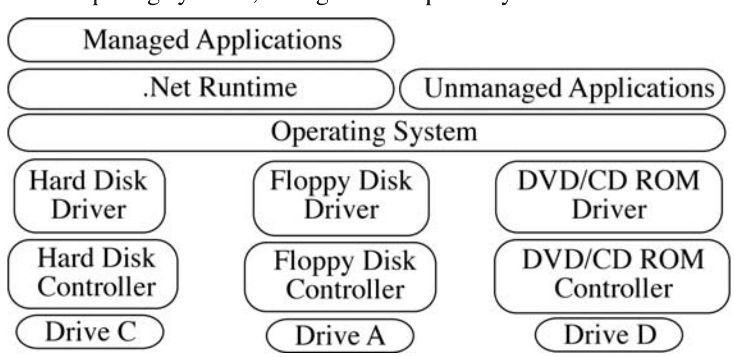


Figure 17-1: Simplified View of Service Layers

Referring to Figure 17-1 — attached storage devices interact with the operating system via an associated software interface referred to as a *driver*. Each device will have its own particular software driver that must be installed and recognized by the operating system before it will work. This applies not only to storage devices but to network cards, display devices, printers, etc. The operating system dictates the rules by which attached storage devices must play, and it is the responsibility of the storage device manufacturer to implement these rules in the device driver.

The operating system makes the services offered by its various device drivers available to running applications. Well-behaved applications target the operating system and do not directly interact with attached storage devices. (**Note:** .NET applications target the .NET runtime environment.)

Files, Directories, And Paths

The Microsoft Windows operating system assigns each attached storage device a letter. On computers with only one hard drive, the letter assigned is ‘C’ and is referred to as your “C drive”. If you have a 3.5 inch floppy drive, its assigned letter is ‘A’. The operating system assigns the next available letter to the next available storage device. Thus, if you also have a CD/ROM or DVD drive, its letter will most likely be ‘D’. If you plug in a removable USB drive, the operating system will assign to it the letter ‘E’ for as long as it’s attached to the machine.

The file, from the operating system’s point of view, is the fundamental storage organizational element. An application’s associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a subdirectory. In modern operating systems like Windows or Apple’s OS X, the metaphors *folder* and *subfolder* are used to refer to a directory and a subdirectory respectively.

The topmost directory structure on a storage device is referred to as the *root* directory. A particular drive’s root directory is indicated by the name of the drive followed by a colon ‘:’, followed by a backward slash character ‘\’. The root directory of the C drive would be “C:\”. Figure 17-2 illustrates these concepts.

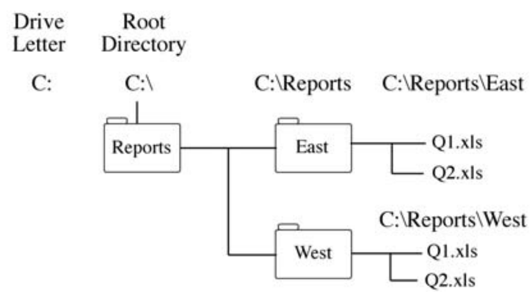


Figure 17-2: Typical Directory Structure

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file’s location can be *absolute* or *relative*. An absolute path includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file’s location. For example, referring to Figure 17-2 — the absolute path to the Microsoft Excel spreadsheet file named Q2.xls located in the East directory, which is located in the Reports directory, which is located in the root directory of the C drive would be:

“`C:\Reports\East\Q2.xls`”.

Figure 17-3 illustrates the concept of an absolute path.

A relative path is the path to a file from some arbitrary starting point, usually a working directory.

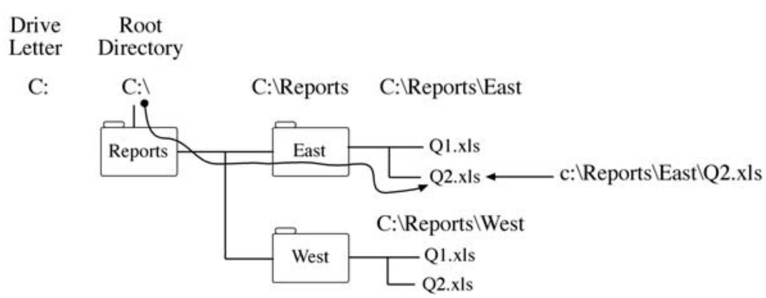


Figure 17-3: The Absolute Path to the Reports\East\Q2.xls File

Manipulating Directories And Files

You can easily create and manipulate directories and files with the help of several classes provided by the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes. The difference between the Directory/File classes vs. DirectoryInfo/FileInfo classes is that the former are static classes while the latter are non-static, meaning you can create instances of FileInfo and DirectoryInfo. Use the

static class versions when you need to perform one or two operations on a directory or file. If you need to do more robust directory or file processing use the -Info versions.

The use of these classes is fairly straightforward. Example 17.1 offers a short program that prints out information about the current directory, the files it contains, and the drives available on the computer.

17.1 DirectoryClassDemo.cs

```
1   using System;
2   using System.IO;
3
4   public class DirectoryClassDemo {
5     public static void Main(){
6       Console.WriteLine("The full path name of the current directory is...");
7       Console.WriteLine("\t" + Directory.GetCurrentDirectory());
8       Console.WriteLine("The current directory has the following files...");
9       String[] files = Directory.GetFiles(Directory.GetCurrentDirectory());
10      foreach(String s in files){
11        FileInfo file = new FileInfo(s);
12        Console.WriteLine("\t" + file.Name);
13      }
14      Console.WriteLine("The computer has the following attached drives...");
15      String[] drives = Directory.GetLogicalDrives();
16      foreach(String s in drives){
17        Console.WriteLine("\t" + s);
18      }
19    }
20  }
```

Referring to Example 17.1 — this example actually demonstrates the use of both the static Directory class and the non-static FileInfo class. On line 7, the Directory.GetCurrentDirectory() method is used to get the absolute path to the current, or working, directory. (*i.e.,* The directory in which the program executes.) On line 9, the Directory.GetFiles() method returns an array of strings representing each of the files in the current working directory. (**Note:** The Directory.GetFileSystemEntries() method would return a string array with the names of all files and directories in the current working directory.)

Given the array of filename strings, the `foreach` statement on line 10 iterates over each entry, creates a new FileInfo object for each filename, and prints its name in the console. You could have simply printed out the array of strings, but that would give you the complete path name of each file. The FileInfo.Name property only returns the name of the file, not its complete path name.

Finally, on line 15, the Directory.GetLogicalDrives() method returns a string array containing the names of all drives connected to the computer. Figure 17-4 shows the results of running this program.

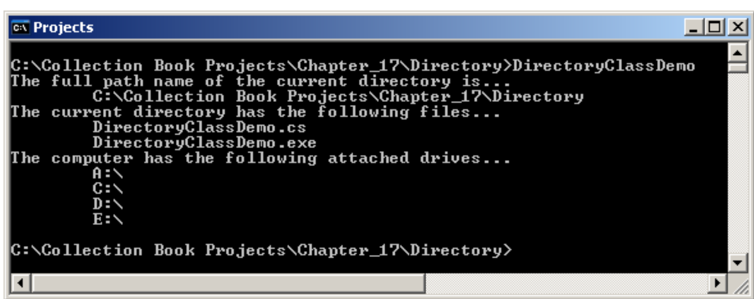


Figure 17-4: Results of Running Example 17.1

Verbatim String Literals

From now on, you will find it more convenient to use *verbatim string literals* rather than ordinary strings when formulating path names. When using ordinary strings, you must precede special characters with the escape character '\'. For example, a path name formulated as an ordinary string would look like this:

```
String path = "c:\\Reports\\East\\Q1.xls"; //ordinary string
```

Verbatim strings are formulated by preceding the string with the '@' character, which signals the compiler to "...interpret the following string literally, including special characters and line breaks." The path string given above would look like this as a verbatim string:

```
String path = @"c:\Reports\East\Q1.xls"; // verbatim string
```


Quick Review

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of data in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When you add a new storage device to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory*.

The topmost directory structure is referred to as the root directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

Verbatim strings are formulated by preceding the string with the '@' character which signals the compiler to "...interpret the following string literally, including special characters and line breaks."

Serializing Objects To Disk

The easiest way to save data to a file is via *serialization*. Serialization is the term used to describe the process of encoding objects in such a way as to facilitate their transmission out of the computer and into or onto some other type of media. Objects can be serialized to disk and then later *deserialized* and reconstituted into objects. The same objects can be serialized for transmission across a network and deserialized at the other end.

While powerful and convenient for you the programmer, serialization is the least flexible way to store data to disk because doing so ties you to the .NET platform. You can't edit the resulting data file. Well, you could edit the file, but because object information is encoded, it's not an ordinary text file, so it's highly likely that you'd screw something up if you did try to edit the file with, say, an ordinary text editor. One way around this is to serialize objects into an XML file.

The nice thing about serialization is that you can serialize single objects, or collections of objects. In this section I will show you how to serialize collections of objects using ordinary serialization with the help of the BinaryFormatter class, and XML serialization with the help of the XMLSerializer class.

Serializable Attribute

Before any object can be serialized it must be tagged as being serializable. You do this by tagging the class with the Serializable attribute. When dealing with collections of objects, not only must the collection itself be serializable — all the objects contained within the collection must be serializable as well. However, you need not worry about collections, and this includes arrays, as they are already tagged as being serializable. Example 17.2 demonstrates the use of the Serializable attribute to make the Dog class serializable.

17.2 Dog.cs

```
using System;

[ Serializable]
public class Dog {

   private String name = null;
   private DateTime birthday;
```



```
8
9      public Dog(String name, DateTime birthday){
10          this.name = name;
11          this.birthday = birthday;
12     }
13
14     public Dog():this("Dog Joe", new DateTime(2005,01,01)){ }
15
16     public Dog(String name):this(name, new DateTime(2005,01,01)){ }
17
18
19     public int Age {
20       get {
21         int years = DateTime.Now.Year - birthday.Year;
22         int adjustment = 0;
23         if(DateTime.Now.Month < birthday.Month){
24             adjustment = 1;
25         }else if((DateTime.Now.Month == birthday.Month) && (DateTime.Now.Day < birthday.Day)){
26                adjustment = 1;
27             }
28         return years - adjustment;
29       }
30    }
31
32     public DateTime Birthday {
33      get { return birthday; }
34      set { birthday = value; }
35
36     }
37
38     public String Name {
39      get { return name; }
40      set { name = value; }
41     }
42
43
44     public override String ToString(){
45       return (name + "," + Age);
46     }
47
48  } // end class definition
```

Referring to Example 17.2 — the Serializable attribute appears on line 3 just above the start of the class definition in square brackets. That's it! This tells the compiler that instances of the Dog class can be serialized. In the next section I'll show you how to serialize an array of Dog objects with the help of the BinaryFormatter class.

Serializing Objects With BinaryFormatter

To serialize an object to disk, you'll need to perform the following steps:

Step 1: Create a FileStream object with the name of the file you want to create on disk.

Step 2: Create a BinaryFormatter object and call its Serialize() method, passing in a reference to a FileStream object and a reference to the object you want to serialize.

Deserialization is the opposite of serialization. Deserialization is the process of reconstituting an object that has been previously serialized and turning it back into an object. To deserialize an object from disk, you must perform the following steps:

Step 1: Create a FileStream object that opens the file that contains the object you want to deserialize.

Step 2: Create a BinaryFormatter object and call its Deserialize() method passing in a reference to the FileStream object.

Step 3: The BinaryFormatter.Deserialize() method returns an object. This object must be cast to the appropriate type.

Example 17.3 offers a short program that serializes and deserializes an array of Dog objects. This program depends on the Dog class presented in Example 17.2.

17.3 MainApp.cs

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;

public class MainApp {
 public static void Main(String[] args){
```

```
8      /***********************************************
9        Create an array of Dogs and populate
10     **********************************************/
11     Dog[] dog_array = new Dog[3];
12
13     dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
14     dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
15     dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
16
17     /*********************************************
18      Iterate over the dog_array and print values
19   *********************************************/
20     Console.WriteLine("-----Original Dog Array Contents---------------------");
21     for(int i = 0; i<dog_array.Length; i++){
22       Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
23     }
24
25     /***********************************************
26      Serialize the array of dog objects to a file
27   *********************************************/
28     FileStream fs = null;
29     try{
30       fs = new FileStream("DogFile.dat", FileMode.Create);
31       BinaryFormatter bf = new BinaryFormatter();
32       bf.Serialize(fs, dog_array);
33
34     }catch(IOException e){
35       Console.WriteLine(e.Message);
36     }catch(SerializationException se){
37       Console.WriteLine(se.Message);
38     }finally{
39        fs.Close();
40   }
41
42   /**********************************************
43      Deserialize the array of dogs and print values
44   ***********************************************/
45        fs = null;                          //start fresh
46        Dog[] another_dog_array = null;     //here too!
47        try{
48          fs = new FileStream("DogFile.dat", FileMode.Open);
49          BinaryFormatter bf = new BinaryFormatter();
50          another_dog_array = (Dog[])bf.Deserialize(fs);
51          Console.WriteLine("-----After Serialization and Deserialization---------");
52          for(int i = 0; i<another_dog_array.Length; i++){
53           Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
54          }
55
56        }catch(IOException e){
57
58        Console.WriteLine(e.Message);
59           }catch(SerializationException se){
60             Console.WriteLine(se.Message);
61           }finally{
62              fs.Close();
63      }
64   } // end Main() definition
65  } // end MainApp class definition
```

Referring to Example 17.3 — note the namespaces you must use to serialize objects to disk with a BinaryFormatter. These include System.IO, System.Runtime.Serialization, and System.Runtime.Serialization.Formatters.Binary. The first thing the program does is create an array of Dogs on line 11 and populate it with references to three Dog objects. The `for` loop starting on line 21 iterates over the dog_array and prints each dog's name and age to the console. The serialization process starts on line 28 with the declaration of the FileStream reference named fs. In the body of the `try` block that begins on line 29, the FileStream object is created using the filename "DogFile.dat" and a FileMode of Create. (**Note:** You can name your files anything you like within the rules of the operating system.)

The BinaryFormatter is created on line 31 and on the next line the Serialize() method is called passing in the reference to the FileStream (fs) and the reference to the array of dogs (dog_array). The appropriate exceptions are handled should something go wrong.

The deserialization process begins on line 45 by setting the reference fs to null and creating a completely new array to house the deserialized array of Dog objects. On line 48, a new FileStream object is created given the appropriate file name and a FileMode of Open. A new BinaryFormatter object is created on the following line and its Deserialize() method is called passing in a reference to the FileStream object. Note how the deserialized object is cast to an

array of Dogs (*i.e. Dog[]*). The `for` loop on line 52 iterates over another_dog_array and prints each dog's name and age to the console. Figure 17-5 shows the results of running this program.

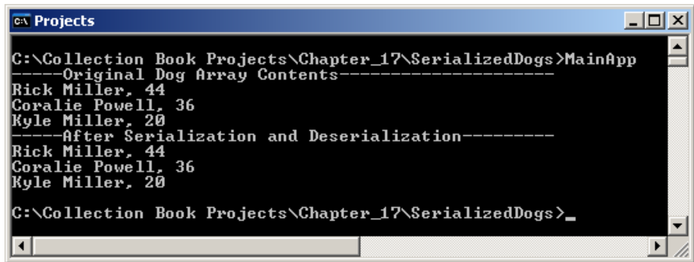


Figure 17-5: Results of Running Example 17.3

Serializing Objects With XMLSerializer

You can serialize objects to disk in XML format with the help of the XMLSerializer class. The steps required to serialize objects to an XML file are similar to those of ordinary serialization:

Step 1: Create a StreamWriter object passing in the name of the file where you want to save the object.

Step 2: Create an XMLSerializer object and call its Serialize() method passing in a reference to the file and to the object you want to serialize.

To deserialize an XML file you would do the following:

Step 1: Create a FileStream object passing in the name of the file you want to read.

Step 2: Create an XMLSerializer object and call its Deserialize() method.

Step 3: The Deserialize() method returns an object. You must cast this object to the appropriate type.

Example 17.4 gives a modified version of MainApp.cs that serializes an array of Dog objects to disk in an XML file.

17.4 MainApp.cs (Mod 1)

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class MainApp {
 public static void Main(String[] args){
   /************************************************
     Create an array of Dogs and populate
   ************************************************/
   Dog[] dog_array = new Dog[3];

   dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
   dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
   dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));

   /***********************************************
    Iterate over the dog_array and print values
  **********************************************/
   Console.WriteLine("-----Original Dog Array Contents---------------------");
   for(int i = 0; i<dog_array.Length; i++){
     Console.WriteLine(dog_array[i].Name + ", " + dog_array[i].Age);
   }

   /*************************************************
    Serialize the array of dog objects to a file
  ***********************************************/
   TextWriter writer = null;
   try{
     writer = new StreamWriter("dogfile.xml");
     XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
     serializer.Serialize(writer, dog_array);


   }catch(IOException ioe){
     Console.WriteLine(ioe.Message);
```



```
37     } catch(Exception ex){
38       Console.WriteLine(ex.Message);
39     } finally{
40        writer.Close();
41      }
42
43   /***********************************************
44      Deserialize the array of dogs and print values
45   ***********************************************/
46        FileStream fs = null;                          //start fresh
47        Dog[] another_dog_array = null;     //here too!
48        try{
49          fs = new FileStream("dogfile.xml", FileMode.Open);
50          XmlSerializer serializer = new XmlSerializer(typeof(Dog[]));
51          another_dog_array = (Dog[])serializer.Deserialize(fs);
52          Console.WriteLine("-----After Serialization and Deserialization---------");
53          for(int i = 0; i<another_dog_array.Length; i++){
54        Console.WriteLine(another_dog_array[i].Name + ", " + another_dog_array[i].Age);
55          }
56
57        } catch(IOException ioe){
58
59        Console.WriteLine(ioe.Message);
60           } catch(Exception ex){
61             Console.WriteLine(ex.Message);
62           } finally{
63              fs.Close();
64      }
65   }  // end Main() definition
66  }  // end MainApp class definition
```

Referring to Example 17.4 — note now that the namespaces required to serialize objects to an XML file include System.IO, System.XML, and System.XML.Serialization. The serialization process begins on line 28 with the declaration of a TextWriter reference. In the body of the `try` block, a StreamWriter object is actually created passing in the name of the file that will be used to hold the serialized dog_array. On line 31, an XMLSerializer object is created. Note that what gets passed as an argument to the constructor is the type of object that will be serialized. The Serialize() method is called on the following line passing in the reference to the output file (writer) and the object to be serialized (dog_array).

The deserialization process starts on line 46 with the declaration of the FileStream reference fs. Another dog array is declared named another_dog_array. In the body of the `try` block starting on line 48, the FileStream object is created passing in the name of the input file and a FileMode of Open. Next, an XMLSerializer object is created again passing to its constructor the type of object that will be deserialized. Lastly, the Deserialize() method is called passing in the name of the input file. The resulting object must be cast to the type Array of Dog (Dog[]). The `for` loop then iterates over the contents of another_dog_array and prints the name and age of each dog to the console. Figure 17-6 gives the results of running this program.

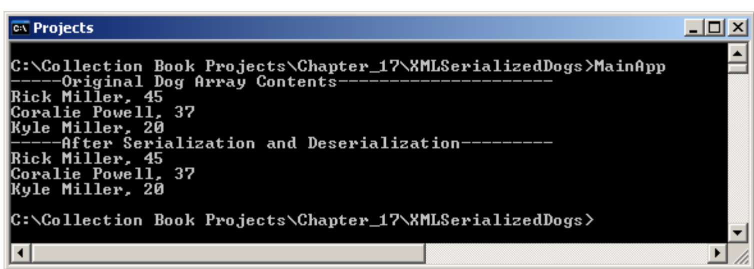


Figure 17-6: Results of Running Example 17.4

At this point you'll find it interesting to explore the contents of both the DogFile.dat and the dogfile.xml files. The DogFile.dat file appears to contain a lot of gibberish, while the XML file is a readable text file that contains XML tags corresponding to the object or objects that were serialized. Example 17.5 gives the listing of dogfile.xml.

17.5 Contents of dogfile.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfDog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <Dog>
    <Birthday>1965-07-08T00:00:00</Birthday>
    <Name>Rick Miller</Name>
  </Dog>
```



```
7       <Dog>
8         <Birthday>1973-08-10T00:00:00</Birthday>
9         <Name>Coralie Powell</Name>
10      </Dog>
11      <Dog>
12        <Birthday>1990-05-01T00:00:00</Birthday>
13        <Name>Kyle Miller</Name>
14      </Dog>
15    </ArrayOfDog>
```

Referring to example 17.5 — I've let line 2 wrap around to the next line due to its length. As you can see, line 2 identifies the type of object which in this case is ArrayOfDog. Each pair of opening and closing <Dog></Dog> tags contains the properties associated with each Dog object.

Quick Review

Object *serialization* provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the Serializable attribute. Place the Serializable attribute above the class declaration line.

When serializing a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

Working With Text Files

One of the best ways to store data in a way that can be easily shared between different applications or different computer platforms is in a *text file*. The System.IO namespace provides two classes that make it easy to process text files: *StreamReader* and *StreamWriter*. The StreamReader class extends the abstract *TextReader* class; the StreamWriter extends the abstract *TextWriter* class.

Some Issues You Must Consider

Before you start writing code to process text files, you'll need to spend some time in the design phase working on exactly what format the text within your text file will have. By format I mean how the text is organized within the file. The decisions you make regarding this issue will vary according to your application's data storage needs. For example, a small database application might store records as separate lines of text. These lines may be, and usually are, separated by special characters referred to as *carriage-return/line-feed* (\r\n). Individual fields within each record may be further separated or *delimited* with another type of character. One character that's commonly used to delimit fields is the comma ','.

Another critically important point to consider is, "What data needs to be preserved in the text file?" For example, if you are working with Person objects within your program, and you want to save this data to a file, what data about each Person object must you save to allow the creation of Person objects later when the data is read from the file?

Also, how might the data be treated later in its life? Will it be read by another program? If so, what type of application is it and how will the data's format affect the application's performance.

Saving Dog Data To A Text File

Example 17.6 offers a short program that saves the data for an array of Dog objects to a text file. After the file is written, the program reads and parses the text file and recreates the array of Dog objects.

17.6 TextFileDemo.cs

```
1   using System;
2   using System.IO;
3
4   public class TextFileDemo {
5     public static void Main(){
6      /**********************************************
7          Create an array of Dogs and populate
8      *********************************************/
9      Dog[] dog_array = new Dog[3];
10
11     dog_array[0] = new Dog("Rick Miller", new DateTime(1965, 07, 08));
12     dog_array[1] = new Dog("Coralie Powell", new DateTime(1973, 08, 10));
13     dog_array[2] = new Dog("Kyle Miller", new DateTime(1990, 05, 01));
14
15     /**********************************************
16      Iterate over the dog_array and print values
17  *********************************************/
18     Console.WriteLine("-----Original Dog Array Contents---------------------");
19     foreach(Dog d in dog_array){
20       Console.WriteLine(d.Name + ", " + d.Age);
21     }
22
23     /*******************************************
24      Save data to textfile
25  *******************************************/
26     TextWriter writer = null;
27     try{
28       writer = new StreamWriter("dogfile.txt");
29       foreach(Dog d in dog_array){
30        writer.WriteLine(d.Name + "," + d.Birthday.Year + "-" + d.Birthday.Month + "-" + d.Birthday.Day);
31       }
32       writer.Flush();
33     }catch(Exception e){
34       Console.WriteLine(e);
35     }finally{
36       writer.Close();
37     }
38
39     /**********************************************
40      Read data from text file and create objects...
41  *********************************************/
42     TextReader reader = null;
43     Dog[] another_dog_array = new Dog[3];
44     try{
45       reader = new StreamReader("dogfile.txt");
46       String s = String.Empty;
47       int count = 0;
48       while((s = reader.ReadLine()) != null){
49         String[] line = s.Split(',');
50         String name = line[0];
51         String[] dob = line[1].Split('-');
52         another_dog_array[count++] = new Dog(name, new DateTime(Int32.Parse(dob[0]), Int32.Parse(dob[1]),
53                                          Int32.Parse(dob[2])));
54       }
55     }catch(Exception e){
56       Console.WriteLine(e);
57     }finally{
58       reader.Close();
59     }
60
61      Console.WriteLine("-----------After writing to and reading from text file------------");
62      foreach(Dog d in another_dog_array){
63         Console.WriteLine(d.Name + ", " + d.Age);
64      }
65
66    } // end Main()
67  } // end class definition
```

Referring to Example 17-6 — the array of Dog references is created as before and each dog's name and age is printed to the console. The start of the text file save process begins on line 26 with the declaration of the TextWriter reference named writer. In the body of the `try` block, a new StreamWriter is created passing in the name of the file in which to save the Dog object data. (dogfile.txt) The `foreach` loop iterates over each element of the array and calls the writer.WriteLine() method to write each dog's name and birthday information to disk. Note that in this case I am separating the name field from the birthday field with a comma.

To create a DateTime object later when I read the file, I will need to have the year, month, and day of the dog's birthday. I delimit each piece of the birthday with a hyphen '-'. When I have finished writing all the lines, I call the writer.Flush() method to actually write the data to disk.

The file read process begins on line 42 with the declaration of a TextReader reference. In the body of the `try` block, I create a StreamReader object passing in the name of the text file to read. I then process the text file according to the following algorithm:

- Declare a string variable in which will be stored each line as it is read from the text file.
- Declare a count variable to control the process loop.
- Read the next line of the file and if it's not null, process the line like so:
    - Declare a string array to hold the individual fields of the string when it is split.
    - Call the String.Split() method to split the line into tokens based on the field delimiter ','.
    - Create a string variable called "name" and assign to it the first token of the split string.
    - Create another string array named dob (short for *date of birth*) to hold the split date field.
    - Call the String.Split() method on the second line token (i.e., line[1]) to split the dob.
    - Create the Dog object using the extracted fields.

As you can see, there is considerably more work involved with manipulating lines of text files. Figure 17-7 gives the results of running this program. Example 17.7 shows the contents of the dogfile.txt file.

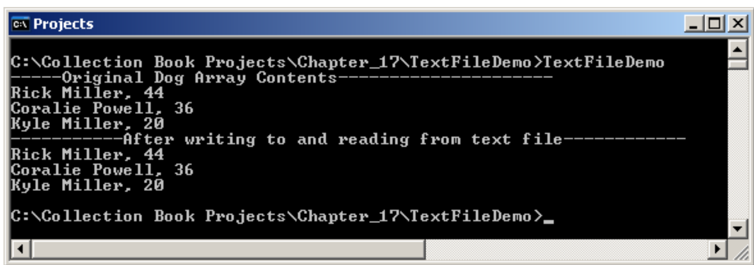


Figure 17-7: Results of Running Example 17.6

17.7 Contents of dogfile.txt

```
Rick Miller,1965-7-8
Coralie Powell,1973-8-10
Kyle Miller,1990-5-1
```

Quick Review

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ',' is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You'll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into tokens with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

Working With Binary Data

You can read and write binary data to a file with the help of the *BinaryReader* and *BinaryWriter* classes. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

Example 17.8 shows the BinaryWriter and BinaryReader classes in action.

17.8 BinaryDataDemo.cs

```
1   using System;
2   using System.IO;
3
4   public class BinaryDataDemo {
5     public static void Main(){
6
7        int record_count = 5;
8        int record_number = 0;
9        int int_val = 125;
10       double double_val = -4567.00;
11       String string_val = "I love C#!";
12       bool bool_val = true;
13
14      /********************************************************
15       Create the file and write the data with a BinaryWriter
16      ******************************************************/
17       BinaryWriter writer = null;
18        try{
19         writer = new BinaryWriter(File.Open("binaryfile.dat", FileMode.Create));
20         writer.Write(record_count);
21         for(int i=0; i<record_count; i++){
22           writer.Write(++record_number);
23           writer.Write(int_val);
24           writer.Write(double_val);
25           writer.Write(string_val);
26           writer.Write(bool_val);
27         }
28
29        }catch(Exception e){
30          Console.WriteLine(e);
31        }finally{
32          writer.Close();
33        }
34
35        /******************************************************
36        Open the file and read the data with a BinaryReader
37       ****************************************************/
38       BinaryReader reader = null;
39       record_count = 0; // reset record count
40       try{
41         reader = new BinaryReader(File.Open("binaryfile.dat", FileMode.Open));
42         record_count = reader.ReadInt32();
43         for(int i=0; i<record_count; i++){
44           Console.WriteLine("Record #:     " + reader.ReadInt32());
45           Console.WriteLine("Int value:    " + reader.ReadInt32());
46           Console.WriteLine("Double value: " + reader.ReadDouble());
47           Console.WriteLine("String value: " + reader.ReadString());
48           Console.WriteLine("Bool value:   " + reader.ReadBoolean());
49           Console.WriteLine("-------------------------------------------------------");
50         }
51
52       }catch(Exception e){
53         Console.WriteLine(e);
54       }finally{
55         reader.Close();
56       }
57    } // end Main()
58  } // end class definition
```

Referring to Example 17.8 — on lines 7 through 12 I declare a set of variables of various different types. I use the variable named record_count to indicate the number of records I'll be writing to and reading from the file. The variable named record_number is incremented for each record that is written to the file and will thus be different for each record. The rest of the variables remain unchanged for the duration of the program.

The BinaryWriter reference named writer is declared on line 17 and is used to write the various simple-type variable values to a file named binaryfile.dat. The `for` loop starting on line 21 writes five records to the file. In this case the boundary of each record, or set of binary values, is demarcated only by the combined length of data written to the file during each iteration of the `for` loop. Also, in this case, the combined length of data written to the file with each iteration of the `for` loop is constant because I don't modify the length of the string variable. If I did, then you'd have variable length records.

The BinaryReader reference named reader is declared on line 38 and is used to read the binary values from the file. How does the reader object know where to read? This is where the concept of a *file position pointer* comes into play. The file position pointer is a variable within the reader object that keeps track of the start of the next read location. It is advanced to the next location based on the length of the type that was just read. For example, if you read an

integer value, the file position pointer is advanced 4 bytes. If the next value read is a string, the pointer is advanced to a point equal to the length of the string. That's why it's important to know exactly what type you are reading and where in the file you are reading it from. In the case of Example 17.8 above, the `for` loop starting on line 43 simply reads the values from the file in the order in which they were written. Figure 17-8 shows the results of running this program.

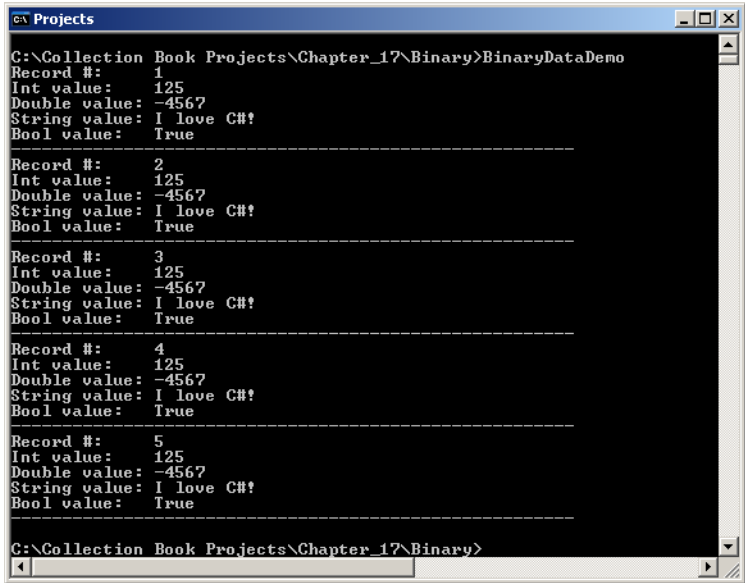


Figure 17-8: Results of Running Example 17.8

Quick Review

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

Random Access File I/O

You can conduct *random access file operations* with the help of the *BinaryReader*, *BinaryWriter*, and *FileStream* classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

There are many ways to go about random access file operations, but generally speaking, you must know a little something about how data is organized in a file so that you know where to find what you are looking for. When seeking a specific record location, you must know where one record ends and another begins. This is not the same as reading lines of text where line terminators provide clues as to where one line ends and a new one begins. In most random access file situations, record length is fixed. (*i.e.,* fixed-length records) A fixed-length record can contain a mixture of binary and character data, but each field within the record is a known size. Seeking the location of a particular record within the file requires the setting of the file position pointer value to a multiple of the record length. The number of records a file contains can be calculated by dividing the file length in bytes by the record length in bytes. You could, of course, randomly seek to any position in a file, but who knows what data you will find there!

In this section I'm going to show you a rather extended example of random access file operations. The example code and resulting application provides a solution to the legacy datafile adapter project specification given in Figure 17-9. Please take some time now to review the project specification before proceeding to the next section.

Towards An Approach To The Adapter Project

Given the project specification and the three supporting artifacts, you may be wondering where to begin. I recommend devoting some time to studying the schema definition and compare it to what you see in the example data file. You will note that although some of the text appears to read OK, there are a few characters here and there that seem out of place. For instance, you can make out the header information, but the header appears to start with a letter ‘z’. Studying the schema definition closely you note that the data file begins with a two-byte file identifier number. But what’s the value of this number?

```
                         Legacy Datafile Adapter
                          Project Specification

Objectives:
  - Demonstrate your ability to conduct random access file I/O operations using the BinaryReader, Binary-
    Writer, and FileStream classes
  - Demonstrate your ability to implement a non-trivial interface
  - Demonstrate your ability to translate low-level exceptions into higher-level, user-defined, applica-
    tion-specific exception abstractions
  - Demonstrate your ability to coordinate file I/O operations via object synchronization

Tasks:
  - You are a junior programmer working in the IT department of a retail bookstore. The CEO wants to begin
    migrating legacy systems to the web using .NET technology. A first step in this initiative is to create
    C# adapters to existing legacy data stores. Given an interface definition, example legacy data file,
    and legacy data file schema definition, write a C# class that serves as an adapter object to a legacy
    data file.

Given:
  - C# interface file specifying adapter operations
  - Legacy data file schema definition
  - Example legacy data file


Legacy Data File Schema Definition:

The legacy data file contains three sections:

1) The file identification section is a two-byte value that identifies the file as a data file.

2) The schema description section immediately follows the first section and contains the field text name
   and two-byte field length for each field in the data section.

3) The data section contains fixed-field-length record data elements arranged according to the following
   schema: (length is in bytes)

   Field Name      Length     Description

   --------------|---------|---------------------------------------------------
   deleted       |    1    |  numeric - 0 if valid, 1 if deleted
   --------------|---------|---------------------------------------------------
   title         |   50    |  text - book title
   --------------|---------|---------------------------------------------------
   author        |   50    |  text - author full name
   --------------|---------|---------------------------------------------------
   pub_code      |    4    |  numeric - publisher code
   --------------|---------|---------------------------------------------------
   ISBN          |   13    |  text - International Standard Book Number
   --------------|---------|---------------------------------------------------
   price         |    8    |  text - retail price in following format: $nnnn.nn
   --------------|---------|---------------------------------------------------
   qoh           |    4    |  numeric - quantity on hand
   --------------|---------|---------------------------------------------------
```

Figure 17-9: Legacy Datafile Adapter Project Specification

Start Small And Take Baby Steps

One way to find out is to write a short program that reads the first two bytes of the file and converts it to a number. The BinaryReader class has a method named ReadInt16(). The method name derives from the System.Int16 structure that represents the short data type in the .NET Framework. A short is a two-byte value. The ReadInt16() method would be an excellent method to use to read the first two bytes of the file in an effort to determine their value.

The next phase of your discovery would be to try and read the rest of the file, or at least try and read the complete header and one complete record using the schema definition as a guide. You may find that a more detailed analysis of the header and record lengths are in order. Figure 17-10 shows a simple analysis performed with a spreadsheet.

	A	B	C	D	E	F	G
2	Header	Section 1	Magic Cookie	2			
3							
4		Section 2	deleted	7			
5			*field length*	2			
6			title	5			
7			*field length*	2			
8			author	6			
9			*field length*	2			
10			pub_code	8			
11			*field length*	2			
12			ISBN	4			
13			*field length*	2			
14			price	5			
15			*field length*	2			
16			qoh	3			
17			*field length*	2			
18							
19			Total Header Length	54			
20							
21							
22	Data		Field Name	Length in Bytes		Offset Start	Offset End
23			deleted	1		0	1
24			title	50		1	51
25			author	50		51	101
26			pub_code	4		101	105
27			ISBN	13		105	118
28			price	8		118	126
29			qoh	4		126	130
30							
31			Total Record Length	130			

Figure 17-10: Header and Record Length Analysis

Referring to Figure 17-10 — the simple analysis reveals that the length of the header section of the legacy data file is 54 bytes long and each record is 130 bytes long. These figures, as well as the individual field lengths, will come in handy when you write the adapter.

Armed with some knowledge about the structure of the legacy data file and having gained some experience writing a small test program that reads all or portions of the file, you can begin to create the adapter class incrementally. A good method to start with is the ReadRecord() method specified in the LegacyDatafileInterface.

Other Project Considerations

This section briefly discusses additional issues which must be considered during the project implementation phase. These considerations include 1) record locking during updates and deletes, and 2) translating low-level I/O exceptions into higher level exceptions as specified in the interface.

Locking A Record For Updates And Deletes

The LegacyDatafileInterface specifies that a record must be locked when it is being updated or deleted. The locking is done via a lock token, which is nothing more that a long value. How might the locking mechanism be implemented? How is the lock_token generated?

To implement the locking mechanism, you must thoroughly understand threads and thread synchronization. (These topics are covered in detail in chapters 13 and 14.) An object can be used as a synchronization point by using the C# `lock` keyword or the Monitor.Enter() and Monitor.Exit() methods. The adapter must ensure that if one thread attempts to update or delete a record (by calling the UpdateRecord() or DeleteRecord() methods), it cannot do so while another thread is in the process of calling either of those methods.

You can adopt several strategies as a means to an ends here. You can 1) apply the synchronized attribute to the entire method in question (UpdateRecord() and DeleteRecord()) or 2) control access only to the critical section of code within each method. Within the locked block, you implement logic to check for a particular condition. If the condition holds, you can proceed with whatever it is you need to do. If the condition does not hold, you will have to wait until it does by calling the Monitor.Wait() method. The Wait() method blocks the current thread and adds it to a list of threads waiting to get a lock on that object.

Conversely, when a thread has obtained a lock on an object and it concludes its business and is ready to release the lock, it can notify other waiting threads to wake up by calling the Monitor.Pulse() method. I have used the `lock` keyword along with Monitor.Wait() and Monitor.Pulse() methods to synchronize access to critical code sections within the DatafileAdapter class.

Monitor.Enter()/Monitor.Exit() vs. The lock Keyword

The `lock` keyword is equivalent to the Monitor.Enter()/Monitor.Exit() method combination. You certainly could use the Monitor.Enter()/Monitor.Exit() combination to control access to a critical code section, but you must take measures to ensure the Monitor.Exit() method gets called at some point. To do this, Microsoft recommends that you use them within the body of a `try/finally` block. The `lock` keyword automatically wraps the Monitor.Enter() and Monitor.Exit() methods in a `try/finally` block for you. Figure 17-11 shows you how the use of the Monitor.Enter()/Monitor.Exit() methods compares to the use of the `lock` keyword.

```
try {                                          lock(Object){
    Monitor.Enter(Object);

    // critical code section                      // critical code section


   } catch(ArgumentNullException e){           }
       // handle  appropriately
   } finally {
        Monitor.Exit(Object);
     }
```

Figure 17-11: Monitor.Enter()/Monitor.Exit() vs. the lock Keyword

Translating Low-Level Exceptions Into Higher-Level Exception Abstractions

The System.IO package defines several low-level exceptions that can occur when conducting file I/O operations. These exceptions must be handled in the adapter, however, the LegacyDatafileInterface specifies that several higher-level exceptions may be thrown when its methods are called.

To create custom exceptions, extend the Exception class and add any customized behavior required. In your adapter code, you catch and handle the low-level exception when it occurs, repackage the exception within the context of a custom exception, and then throw the custom exception. Any objects utilizing the services of the adapter class must handle your custom exceptions, not the low-level I/O exceptions.

Where To Go From Here

The previous sections attempted to address some of the development issues you will typically encounter when attempting this type of project. The purpose of the project is to demonstrate the use of the FileStream, BinaryReader, and BinaryWriter classes in the context of a non-trivial example. I hope also that I have sufficiently illustrated the reality that rarely can one class perform its job without the help of many other classes.

The next section gives the code for the completed project. Keep in mind that the examples listed here represent one particular approach and solution to the problem. As an exercise, I will invite you to attempt a solution on your own terms using the knowledge gained here as a guide.

Explore and study the code. Compile the code and observe its operation. Experiment — make changes to areas you feel can use improvement.

Complete RandomAccessFile Legacy Datafile Adapter Source Code Listing

This section gives the complete listing for the code that satisfies the requirements of the Legacy Datafile Adapter project.

17.9 FailedRecordCreationException.cs

```
using System;

public class FailedRecordCreationException : Exception {

    public FailedRecordCreationException() : base("Failed Record Creation Exception") { }

    public FailedRecordCreationException(String message) : base(message) { }

    public FailedRecordCreationException(String message, Exception inner_exception) :
                                 base(message, inner_exception) { }
}
```


17.10 InvalidDataFileExcepton.cs

```
using System;

public class InvalidDataFileException : Exception {

    public InvalidDataFileException() : base("Invalid Data File Exception") { }

    public InvalidDataFileException(String message) : base(message) { }

    public InvalidDataFileException(String message, Exception inner_exception) :
                           base(message, inner_exception) { }
}
```

17.11 NewDatafileException.cs

```
using System;

public class NewDataFileException : Exception {

    public NewDataFileException() : base("New Data File Exception") { }

    public NewDataFileException(String message) : base(message) { }

    public NewDataFileException(String message, Exception inner_exception) :
                         base(message, inner_exception) { }
}
```

17.12 RecordNotFoundException.cs

```
using System;

public class RecordNotFoundException : Exception {

    public RecordNotFoundException() : base("Record Not Found Exception") { }

    public RecordNotFoundException(String message) : base(message) { }

    public RecordNotFoundException(String message, Exception inner_exception) :
                          base(message, inner_exception) { }
}
```

17.13 SecurityException.cs

```
using System;

public class SecurityException : Exception {

    public SecurityException() : base("Security Exception") { }

    public SecurityException(String message) : base(message) { }

    public SecurityException(String message, Exception inner_exception) :
                       base(message, inner_exception) { }
}
```

17.14 LegacyDatafileInterface.cs

```
using System;

public interface LegacyDatafileInterface {


  /// <summary>
  /// Read the record indicated by the rec_no and return a string array
  /// were each element contains a field value.
  /// </summary>
  /// <param name="rec_no"></param>
  /// <returns>A string array containing the record fields</returns>
  /// <exception cref="RecordNotFoundException"</exception>
  String[] ReadRecord(long rec_no);


  /// <summary>
  /// Update a record's fields. The record must be locked with the lockRecord()
  /// method and the lock_token must be valid. The value for field n appears in
  /// element record[n].
  /// </summary>
  /// <param name="rec_no"></param>
  /// <param name="record"></param>
  /// <param name="lock_token"></param>
```



```
   /// <exception cref="RecordNotFoundException"></exception>
   /// <exception cref="SecurityException"></exception>
   void UpdateRecord(long rec_no, String[] record, long lock_token);


   /// <summary>
   /// Marks a record for deletion by setting the deleted field to 1. The lock_token
   /// must be valid otherwise a SecurityException is thrown.
   /// </summary>
   /// <param name="rec_no"></param>
   /// <param name="lock_token"></param>
   /// <exception cref="RecordNotFoundException" ></exception>
   /// <exception cref="SecurityException"></exception>
   void DeleteRecord(long rec_no, long lock_token);

   /// <summary>
   /// Creates a new datafile record and returns the record number.
   /// </summary>
   /// <param name="record"></param>
   /// <returns>The record number of the newly created record</returns>
   /// <exception cref="FailedRecordCreationException"></exception>
   long CreateRecord(String[] record);


   /// <summary>
   /// Locks a record for updates and deletes and returns an integer
   /// representing a lock token.
   /// </summary>
   /// <param name="rec_no"></param>
   /// <returns>Lock token</returns>
   /// <exception cref="RecordNotFoundException"></exception>
   long LockRecord(long rec_no);


   /// <summary>
   /// Unlocks a previously locked record. The lock_token must be valid or a
   /// SecurityException is thrown.
   /// </summary>
   /// <param name="rec_no"></param>
   /// <param name="lock_token"></param>
   /// <exception cref="SecurityException"></exception>
   void UnlockRecord(long rec_no, long lock_token);


   /// <summary>
   ///  Searches the records in the datafile for records that match the String
   /// values of search_criteria. search_criteria[n] contains the search value
   /// applied against field n.
   /// </summary>
   /// <param name="search_criteria"></param>
   /// <returns>An array of longs containing the matched record numbers</returns>
   long[] SearchRecords(String[] search_criteria);

 } //end interface definition
```

17.15 DataFileAdapter.cs

```
using System;
using System.IO;
using System.Text;
using System.Threading;
using System.Collections;
using System.Collections.Generic;


public class DataFileAdapter : LegacyDatafileInterface {

    /**************************************
     * Constants
     ****************************************/

    private const short FILE_IDENTIFIER = 378;
    private const int HEADER_LENGTH = 54;
    private const int RECORDS_START = 54;
    private const int RECORD_LENGTH = 130;
    private const int FIELD_COUNT = 7;

    private const short DELETED_FIELD_LENGTH = 1;
    private const short TITLE_FIELD_LENGTH = 50;
    private const short AUTHOR_FIELD_LENGTH = 50;
    private const short PUB_CODE_FIELD_LENGTH = 4;
```

```
    private const short ISBN_FIELD_LENGTH = 13;
    private const short PRICE_FIELD_LENGTH = 8;
    private const short QOH_FIELD_LENGTH = 4;

    private const String DELETED_STRING = "deleted";
    private const String TITLE_STRING = "title";
    private const String AUTHOR_STRING = "author";
    private const String PUB_CODE_STRING = "pub_code";
    private const String ISBN_STRING = "ISBN";
    private const String PRICE_STRING = "price";
    private const String QOH_STRING = "qoh";

    private const int TITLE_FIELD = 0;
    private const int AUTHOR_FIELD = 1;
    private const int PUB_CODE_FIELD = 2;
    private const int ISBN_FIELD = 3;
    private const int PRICE_FIELD = 4;
    private const int QOH_FIELD = 5;

    private const int VALID = 0;
    private const int DELETED = 1;

    /********************************************************
   * Private Instance Fields
   ********************************************************/
    private String _filename = null;
    private BinaryReader _reader = null;
    private BinaryWriter _writer = null;
    private long _record_count = 0;
    private Hashtable _locked_records_map = null;
    private Random _token_maker = null;
    private long _current_record_number = 0;
    private bool _debug = false;

    /*******************************************************
   * Properties
   *****************************************************/
    public long RecordCount {
       get { return _record_count; }
     }

    /********************************************************
   * Instance Methods
   *******************************************************/

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="filename"></param>
    /// <exception cref="InvalidDataFileException"></exception>
    public DataFileAdapter(String filename) {
      try {
        _filename = filename;
        if(File.Exists(_filename)){
          _reader = new BinaryReader(File.Open(filename, FileMode.Open));
          if ((_reader.BaseStream.Length >= HEADER_LENGTH) && (_reader.ReadInt16() == FILE_IDENTIFIER)) {
         // it's a valid data file
             Console.WriteLine(_filename + " is a valid data file...");
             _record_count = ((_reader.BaseStream.Length - HEADER_LENGTH) / RECORD_LENGTH);
             Console.WriteLine("Record count is: " + _record_count);
             InitializeVariables();
             _reader.Close();
            } else if (_reader.BaseStream.Length == 0) {  // The file's empty - make it a data file
                    _reader.Close();
                    WriteHeader(FileMode.Open);
                    InitializeVariables();
                  } else {
                      _reader.BaseStream.Seek(0, SeekOrigin.Begin);
                      if (_reader.ReadInt16() != FILE_IDENTIFIER) {
                        _reader.Close();
                        Console.WriteLine("Invalid data file. Closing file.");
                        throw new InvalidDataFileException("Invalid data file identifier...");
                      }
                  }
        } else {
          CreateNewDataFile(_filename);
        }
      } catch (ArgumentException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("Invalid argument.",e);
        }
```



```
       catch (EndOfStreamException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("End of stream exception.",e);
        }
       catch (ObjectDisposedException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("BinaryReader not initialized.",e);
        }
       catch (IOException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("General IOException",e);
        }
       catch (Exception e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("General Exception",e);
        }
        finally {
          if (_reader != null) {
            _reader.Close();
          }
        }
    } // end constructor


    /// <summary>
    /// Default Constructor
    /// </summary>
    /// <exception cref="InvalidDataFileException"></exception>
    public DataFileAdapter():this("books.dat"){ }


    /// <summary>
    /// Create new file
    /// </summary>
    /// <param name="filename"></param>
    /// <exception cref="NewDataFileException"></exception>
    public void CreateNewDataFile(String filename) {
      try {
        _filename = filename;
        WriteHeader(FileMode.Create);
        InitializeVariables();
      } catch (Exception e) {
          if(_debug) { Console.WriteLine(e); }
          throw new NewDataFileException(e.ToString());
        }
    } // end createNewDataFile method


    /// <summary>
    /// Read the record indicated by the rec_no and return a string array
    /// were each element contains a field value.
    /// </summary>
    /// <param name="rec_no"></param>
    /// <returns>A populated string array containing record field values</returns>
    /// <exception cref="RecordNotFoundException"></exception>
    public String[] ReadRecord(long rec_no) {
      String[] temp_string = null;
      if ((rec_no < 0) || (rec_no > _record_count)) {
        if(_debug){ Console.WriteLine("From ReadRecord(): Requested record out of range!"); }
         throw new RecordNotFoundException("From ReadRecord(): Requested record out of range");
      } else {
          try {
            _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
            GotoRecordNumber(_reader, rec_no);
            if (_reader.ReadByte() == DELETED) {
              if(_debug){ Console.WriteLine("From ReadRecord(): Record number " + rec_no +
                                   " has been deleted!"); }
              throw new RecordNotFoundException("Record " + rec_no + " deleted!");
            } else {
                temp_string = RecordBytesToStringArray(_reader, rec_no);
              }
          } catch (ArgumentException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new RecordNotFoundException("Invalid argument.",e);
        }
       catch (EndOfStreamException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new RecordNotFoundException("End of stream exception.",e);
        }
       catch (ObjectDisposedException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
```

```
          throw new RecordNotFoundException("BinaryReader not initialized.",e);
        }
       catch (IOException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new RecordNotFoundException("General IOException",e);
        }
       catch (Exception e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new RecordNotFoundException("General Exception",e);
        }
            finally {
                if (_reader != null) {
                    _reader.Close();
                }
            }
        } // end else
      return temp_string;
    } // end readRecord()


    /// <summary>
    /// Update a record's fields. The record must be locked with the lockRecord()
    /// method and the lock_token must be valid. The value for field n appears in
    /// element record[n]. The call to updateRecord() MUST be preceeded by a call
    /// to lockRecord() and followed by a call to unlockRecord()
    /// </summary>
    /// <param name="rec_no"></param>
    /// <param name="record"></param>
    /// <param name="lock_token"></param>
    /// <exception cref="RecordNotFoundException"></exception>
    /// <exception cref="SecurityException"></exception>
    public void UpdateRecord(long rec_no, String[] record, long lock_token) {
      if (lock_token != ((long)_locked_records_map[rec_no])) {
        if(_debug){ Console.WriteLine("From UpdateRecord(): Invalid  update record lock token."); }
        throw new SecurityException("From UpdateRecord(): Invalid update record lock token.");
      } else {
          try {
            _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
            GotoRecordNumber(_writer, rec_no); //i.e., goto indicated record
            _writer.Write((byte)0);
            _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
            _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
            _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
            _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
            _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
            _writer.Write(Int16.Parse(record[QOH_FIELD]));
            _current_record_number = rec_no;
          } catch (ArgumentException e) {
             if(_debug){ Console.WriteLine(e.ToString()); }
             throw new RecordNotFoundException("Invalid argument.",e);
            }
           catch (EndOfStreamException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("End of stream exception.",e);
           }
           catch (ObjectDisposedException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("BinaryReader not initialized.",e);
            }
           catch (IOException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("General IOException",e);
           }
           catch (Exception e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("General Exception",e);
           }
            finally {
              if (_writer != null) {
                 _writer.Close();
              }
            }
        }// end else
    }// end updateRecord()


    /// <summary>
    /// Marks a record for deletion by setting the deleted field to 1. The lock_token
    /// must be valid otherwise a SecurityException is thrown.
    /// </summary>
    /// <param name="rec_no"></param>
```



```
    /// <param name="lock_token"></param>
    /// <exception cref="RecordNotFoundException"></exception>
    /// <exception cref="SecurityException"></exception>
    public void DeleteRecord(long rec_no, long lock_token) {
      if (lock_token != (long)_locked_records_map[rec_no]) {
        Console.WriteLine("From DeleteRecord(): Invalid delete record lock token.");
        throw new SecurityException("From DeleteRecord(): Invalid delete record lock token.");
      } else {
          try {
            _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
            GotoRecordNumber(_writer, rec_no); // goto record indicated
            _writer.Write((byte)1);  // mark for deletion
          } catch (ArgumentException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("Invalid argument.",e);
           }
           catch (EndOfStreamException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("End of stream exception.",e);
           }
           catch (ObjectDisposedException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("BinaryReader not initialized.",e);
           }
           catch (IOException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("General IOException",e);
           }
          catch (Exception e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new RecordNotFoundException("General Exception",e);
          }
            finally {
              if (_writer != null) {
                 _writer.Close();
              }
            }
        }// end else
    }// end deleteRecord()


    /// <summary>
    /// Creates a new datafile record and returns the record number.
    /// </summary>
    /// <param name="record"></param>
    /// <returns> The record number of the newly created record</returns>
    /// <exception cref="FailedRecordCreationException"></exception>
    public long CreateRecord(String[] record) {
      try {
        _writer = new BinaryWriter(File.Open(_filename, FileMode.Open));
        GotoRecordNumber(_writer, _record_count); //i.e., goto end of file
        _writer.Write((byte)0);
        _writer.Write(StringToPaddedByteField(record[TITLE_FIELD], TITLE_FIELD_LENGTH));
        _writer.Write(StringToPaddedByteField(record[AUTHOR_FIELD], AUTHOR_FIELD_LENGTH));
        _writer.Write(Int16.Parse(record[PUB_CODE_FIELD]));
        _writer.Write(StringToPaddedByteField(record[ISBN_FIELD], ISBN_FIELD_LENGTH));
        _writer.Write(StringToPaddedByteField(record[PRICE_FIELD], PRICE_FIELD_LENGTH));
        _writer.Write(Int16.Parse(record[QOH_FIELD]));
        _current_record_number = ++_record_count;
      } catch (ArgumentException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new FailedRecordCreationException("Invalid argument.",e);
        }
       catch (EndOfStreamException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new FailedRecordCreationException("End of stream exception.",e);
        }
       catch (ObjectDisposedException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new FailedRecordCreationException("BinaryReader not initialized.",e);
        }
       catch (IOException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new FailedRecordCreationException("General IOException",e);
        }
       catch (Exception e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new FailedRecordCreationException("General Exception",e);
        }
        finally {
          if (_writer != null) {
```



```
            _writer.Close();
          }
        }
      return _current_record_number;
    } // end CreateRecord()


    /// <summary>
    /// Locks a record for updates and deletes  - returns an integer
    /// representing a lock token.
    /// </summary>
    /// <param name="rec_no"></param>
    /// <returns></returns>
    /// <exception cref="RecordNotFoundException"></exception>
    public long LockRecord(long rec_no) {
      long lock_token = 0;
      if ((rec_no < 0) || (rec_no > _record_count)) {
        if(_debug){ Console.WriteLine("Record cannot be locked. Not in valid range."); }
        throw new RecordNotFoundException("Record cannot be locked. Not in valid range.");
      } else {
          lock (_locked_records_map) {
          while (_locked_records_map.ContainsKey(rec_no)) {
            try {
              Monitor.Wait(_locked_records_map);
            } catch (Exception) { }
          }
          lock_token = (long)_token_maker.Next();
          _locked_records_map.Add(rec_no, lock_token);
          } // end lock
        } // end else
      return lock_token;
    } // end LockRecord()


    /// <summary>
    /// Unlocks a previously locked record. The lock_token must be valid or a
    /// SecurityException is thrown.
    /// </summary>
    /// <param name="rec_no"></param>
    /// <param name="lock_token"></param>
    /// <exception cref="SecurityException"></exception>
    public void UnlockRecord(long rec_no, long lock_token) {
      lock (_locked_records_map) {
        if (_locked_records_map.Contains(rec_no)) {
          if (lock_token == ((long)_locked_records_map[rec_no])) {
             _locked_records_map.Remove(rec_no);
             Monitor.Pulse(_locked_records_map);
          } else {
              if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid lock token."); }
              throw new SecurityException("From UnlockRecord(): Invalid lock token");
            }
          } else {
              if(_debug){ Console.WriteLine("From UnlockRecord(): Invalid record number."); }
              throw new SecurityException("From UnlockRecord(): Invalid record number.");
            }
        }
    } // end UnlockRecord()


    /// <summary>
    /// Searches the records in the datafile for records that match the String
    /// values of search_criteria. search_criteria[n] contains the search value
    /// applied against field n. Data files can be searched for Title & Author.
    /// </summary>
    /// <param name="search_criteria"></param>
    /// <returns>An array of long values each indicating a record number match</returns>
    public long[] SearchRecords(String[] search_criteria) {
      List<long> hit_list = new List<long>();
      for (long i = 0; i < _record_count; i++) {
        try {
          if (ThereIsAMatch(search_criteria, ReadRecord(i))) {
             hit_list.Add(i);
          }
        } catch (RecordNotFoundException) { } // ignore deleted records
      } // end for
      long[] hits = new long[hit_list.Count];
      for (int i = 0; i < hits.Length; i++) {
        hits[i] = hit_list[i];
      }
      return hits;
    } // end SearchRecords()
```

```
      /// <summary>
      /// ThereIsAMatch() is a utility method that actually performs
      /// the record search. Implements an implied OR/AND search by detecting
      /// the first character of the Title criteria element.
      /// </summary>
      /// <param name="search_criteria"></param>
      /// <param name="record"></param>
      /// <returns>A boolean value indicating true if there is a match or false otherwise.</returns>
      private bool ThereIsAMatch(String[] search_criteria, String[] record) {
        bool match_result = false;
        int TITLE = 0;
        int AUTHOR = 1;
        for (int i = 0; i < search_criteria.Length; i++) {
          if ((search_criteria[i].Length == 0) || (record[i + 1].StartsWith(search_criteria[i]))) {
             match_result = true;
             break;
          } //end if
        } //end for

        if (((search_criteria[TITLE].Length > 1) && (search_criteria[AUTHOR].Length >= 1)) &&
                                         (search_criteria[TITLE][0] == '&')) {
        if (record[TITLE + 1].StartsWith(search_criteria[TITLE].Substring(1,
                                                search_criteria[TITLE].Length).Trim())
           && record[AUTHOR + 1].StartsWith(search_criteria[AUTHOR])) {
                 match_result = true;
           } else {
               match_result = false;
             }
        } // end  outer if
        return match_result;
      } // end thereIsAMatch()


      /// <summary>
      /// GotoRecordNumber - utility function that handles the messy
      /// details of seeking a particular record.
      /// </summary>
      /// <param name="record_number"></param>
      /// <exception cref="RecordNotFoundException"></exception>
      private void GotoRecordNumber(BinaryReader reader, long record_number) {
        if ((record_number < 0) || (record_number > _record_count)) {
          throw new RecordNotFoundException();
        } else {
            try {
              reader.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
            } catch (EndOfStreamException e) {
                if(_debug){ Console.WriteLine(e.ToString()); }
                throw new RecordNotFoundException("End of stream exception.",e);
              }
              catch (ObjectDisposedException e) {
                if(_debug){ Console.WriteLine(e.ToString()); }
                throw new RecordNotFoundException("BinaryReader not initialized.",e);
              }
              catch (IOException e) {
                if(_debug){ Console.WriteLine(e.ToString()); }
                throw new RecordNotFoundException("General IOException",e);
              }
              catch (Exception e) {
                if(_debug){ Console.WriteLine(e.ToString()); }
                throw new RecordNotFoundException("General Exception",e);
              }
          }// end else
      } // end GotoRecordNumber()


      /// <summary>
      /// GotoRecordNumber - overloaded utility function that handles the messy
      /// details of seeking a particular record.
      /// </summary>
      /// <param name="record_number"></param>
      /// <exception cref="RecordNotFoundException"></exception>
      private void GotoRecordNumber(BinaryWriter writer, long record_number) {
        if ((record_number < 0) || (record_number > _record_count)) {
          throw new RecordNotFoundException();
        } else {
            try {
              writer.BaseStream.Seek(RECORDS_START + (record_number * RECORD_LENGTH), SeekOrigin.Begin);
            } catch (EndOfStreamException e) {
                if(_debug){ Console.WriteLine(e.ToString()); }
```



```
              throw new RecordNotFoundException("End of stream exception.",e);
            }
            catch (ObjectDisposedException e) {
              if(_debug){ Console.WriteLine(e.ToString()); }
              throw new RecordNotFoundException("BinaryReader not initialized.",e);
            }
            catch (IOException e) {
              if(_debug){ Console.WriteLine(e.ToString()); }
              throw new RecordNotFoundException("General IOException",e);
            }
            catch (Exception e) {
              if(_debug){ Console.WriteLine(e.ToString()); }
              throw new RecordNotFoundException("General Exception",e);
            }
        }// end else
    } // end GotoRecordNumber()


    /// <summary>
    /// stringToPaddedByteField - pads the field to maintain fixed
    /// field length.
    /// </summary>
    /// <param name="s"></param>
    /// <param name="field_length"></param>
    /// <returns>A populated byte array containing the string value padded with spaces</returns>
    protected byte[] StringToPaddedByteField(String s, int field_length) {
      byte[] byte_field = new byte[field_length];
      if (s.Length <= field_length) {
         for (int i = 0; i < s.Length; i++) {
           byte_field[i] = (byte)s[i];
         }
         for (int i = s.Length; i < field_length; i++) {
           byte_field[i] = (byte)' '; //pad the field
         }
      } else {
          for (int i = 0; i < field_length; i++) {
            byte_field[i] = (byte)s[i];
          }
        }
      return byte_field;
    } // end StringToPaddedByteField()


    /// <summary>
    /// RecordBytesToStringArray - reads an array of bytes from a data file
    /// and converts them to an array of Strings. The first element of the
    /// returned array is the record number. The length of the byte array
    /// argument is RECORD_LENGTH -1.
    /// </summary>
    /// <param name="record_number"></param>
    /// <returns></returns>
    private String[] RecordBytesToStringArray(BinaryReader reader, long record_number) {
      String[] string_array = new String[FIELD_COUNT];
      char[] title = new char[TITLE_FIELD_LENGTH];
      char[] author = new char[AUTHOR_FIELD_LENGTH];
      char[] isbn = new char[ISBN_FIELD_LENGTH];
      char[] price = new char[PRICE_FIELD_LENGTH];
      try {
        string_array[0] = record_number.ToString();
        reader.Read(title, 0, title.Length);
        string_array[TITLE_FIELD + 1] = new String(title).Trim();
        reader.Read(author, 0, author.Length);
        string_array[AUTHOR_FIELD + 1] = new String(author).Trim();
        string_array[PUB_CODE_FIELD + 1] = (reader.ReadInt16()).ToString();
        reader.Read(isbn, 0, isbn.Length);
        string_array[ISBN_FIELD + 1] = new String(isbn);
        reader.Read(price, 0, price.Length);
        string_array[PRICE_FIELD + 1] = new String(price).Trim();
        string_array[QOH_FIELD + 1] = (reader.ReadInt16()).ToString();
      } catch (IOException e) {
          Console.WriteLine(e.ToString());
        }
      return string_array;
    } // end recordBytesToStringArray()


    /// <summary>
    /// Writes the header information into a data file
    /// </summary>
    /// <exception cref="InvalidDataFileException"></exception>
    private void WriteHeader(FileMode file_mode) {
```



```
      try {
        if (_writer != null) {
          _writer.Close();
        }
        _writer = new BinaryWriter(File.Open(_filename, file_mode));
        _writer.Seek(0, SeekOrigin.Begin);
        _writer.Write(FILE_IDENTIFIER);
        _writer.Write(DELETED_STRING.ToCharArray());
        _writer.Write(DELETED_FIELD_LENGTH);
        _writer.Write(TITLE_STRING.ToCharArray());
        _writer.Write(TITLE_FIELD_LENGTH);
        _writer.Write(AUTHOR_STRING.ToCharArray());
        _writer.Write(AUTHOR_FIELD_LENGTH);
        _writer.Write(PUB_CODE_STRING.ToCharArray());
        _writer.Write(PUB_CODE_FIELD_LENGTH);
        _writer.Write(ISBN_STRING.ToCharArray());
        _writer.Write(ISBN_FIELD_LENGTH);
        _writer.Write(PRICE_STRING.ToCharArray());
        _writer.Write(PRICE_FIELD_LENGTH);
        _writer.Write(QOH_STRING.ToCharArray());
        _writer.Write(QOH_FIELD_LENGTH);
        _writer.Flush();
      } catch (ArgumentException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("Invalid argument.",e);
        }
       catch (EndOfStreamException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("End of stream exception.",e);
        }
       catch (ObjectDisposedException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("BinaryReader not initialized.",e);
        }
       catch (IOException e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("General IOException",e);
        }
       catch (Exception e) {
          if(_debug){ Console.WriteLine(e.ToString()); }
          throw new InvalidDataFileException("General Exception",e);
        }
        finally {
          if (_writer != null) {
            _writer.Close();
          }
        }
    }  // end WriteHeader()


    /// <summary>
    /// readHeader - reads the header bytes and converts them to
    /// a string
    /// </summary>
    /// <returns> A String containing the file header information</returns>
    /// <exception cref="InvalidDataFileException"></exception>
    public String ReadHeader() {
      StringBuilder sb = new StringBuilder();
      char[] deleted = new char[DELETED_STRING.Length];
      char[] title = new char[TITLE_STRING.Length];
      char[] author = new char[AUTHOR_STRING.Length];
      char[] pub_code = new char[PUB_CODE_STRING.Length];
      char[] isbn = new char[ISBN_STRING.Length];
      char[] price = new char[PRICE_STRING.Length];
      char[] qoh = new char[QOH_STRING.Length];
      try {
        _reader = new BinaryReader(File.Open(_filename, FileMode.Open));
        _reader.BaseStream.Seek(0, SeekOrigin.Begin);
        sb.Append(_reader.ReadInt16() + " ");
        _reader.Read(deleted, 0, deleted.Length);
        sb.Append(new String(deleted) + " ");
        sb.Append(_reader.ReadInt16() + " ");
        _reader.Read(title, 0, title.Length);
        sb.Append(new String(title) + " ");
        sb.Append((_reader.ReadInt16()) + " ");
        _reader.Read(author, 0, author.Length);
        sb.Append(new String(author) + " ");
        sb.Append((_reader.ReadInt16()) + " ");
        _reader.Read(pub_code, 0, pub_code.Length);
        sb.Append(new String(pub_code) + " ");
        sb.Append((_reader.ReadInt16()) + " ");
```



```
          _reader.Read(isbn, 0, isbn.Length);
          sb.Append(new String(isbn) + " ");
          sb.Append((_reader.ReadInt16()) + " ");
          _reader.Read(price, 0, price.Length);
          sb.Append(new String(price) + " ");
          sb.Append((_reader.ReadInt16()) + " ");
          _reader.Read(qoh, 0, qoh.Length);
          sb.Append(new String(qoh) + " ");
          sb.Append((_reader.ReadInt16()) + " ");
        } catch (ArgumentException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new InvalidDataFileException("Invalid argument.",e);
          }
         catch (EndOfStreamException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new InvalidDataFileException("End of stream exception.",e);
          }
         catch (ObjectDisposedException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new InvalidDataFileException("BinaryReader not initialized.",e);
          }
         catch (IOException e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new InvalidDataFileException("General IOException",e);
          }
         catch (Exception e) {
            if(_debug){ Console.WriteLine(e.ToString()); }
            throw new InvalidDataFileException("General Exception",e);
          }
          finally {
            if (_reader != null) {
              _reader.Close();
            }
          }
        return sb.ToString();
      } // end ReadHeader()


      /// <summary>
      /// Utility method used to initialize several important instance fields
      /// </summary>
      private void InitializeVariables() {
        _current_record_number = 0;
        _locked_records_map = new Hashtable();
        _token_maker = new Random();
      }

} // end DataFileAdapter class definition
```

17.16 AdapterTestApp.cs

```
using System;

public class AdapterTesterApp {
   public static void Main(){
     try{
       DataFileAdapter adapter = new DataFileAdapter("books.dat");
       String[] rec_1 = {"C++ For Artists", "Rick Miller", "0001", "1-932504-02-8", "$59.95", "80"};
       String[] rec_2 = {"Java For Artists", "Rick Miller", "0002", "1-932504-04-X", "$69.95", "100"};
       String[] rec_3 = { "C# For Artists", "Rick Miller", "0003", "1-932504-07-9", "$76.00", "567" };
       String[] rec_4 = { "White Saturn", "Rick Miller", "0004", "1-932504-08-7", "$45.00", "234" };

       String[] search_string = {"Java", " "};

       String[] temp_string = null;

       adapter.CreateRecord(rec_1);
       adapter.CreateRecord(rec_2);
       adapter.CreateRecord(rec_3);
       adapter.CreateRecord(rec_1);
       adapter.CreateRecord(rec_2);
       adapter.CreateRecord(rec_3);
       adapter.CreateRecord(rec_1);
       adapter.CreateRecord(rec_2);
       adapter.CreateRecord(rec_3);


       long lock_token = adapter.LockRecord(2);

```



```
        adapter.UpdateRecord(2, rec_2, lock_token);
        adapter.UnlockRecord(2, lock_token);

        lock_token = adapter.LockRecord(1);
        adapter.DeleteRecord(1, lock_token);
        adapter.UnlockRecord(1, lock_token);

        lock_token = adapter.LockRecord(4);
        adapter.UpdateRecord(4, rec_4, lock_token);
        adapter.UnlockRecord(4, lock_token);

        long[] search_hits = adapter.SearchRecords(search_string);

        Console.WriteLine(adapter.ReadHeader());

        for(int i=0; i<search_hits.Length; i++){
          try{
          temp_string = adapter.ReadRecord(search_hits[i]);
          for(int j = 0; j<temp_string.Length; j++){
           Console.Write(temp_string[j] + " ");
         }
         Console.WriteLine();
             }catch(RecordNotFoundException){ }
      }

        Console.WriteLine("----------------------------------------");
        for (int i = 0; i < adapter.RecordCount; i++) {
          try {
            temp_string = adapter.ReadRecord(i);
            for (int j = 0; j < temp_string.Length; j++) {
              Console.Write(temp_string[j] + " ");
            }
            Console.WriteLine();
          }
          catch (RecordNotFoundException) {  }
        }
      }
      catch (Exception e) { Console.WriteLine(e.ToString()); }
    } // end Main()
  } // end class definition
```

Figure 17-11 shows the results of running the AdapterTestApp program one time. Running it several times back-to-back results in additional records being inserted into the book.dat data file.

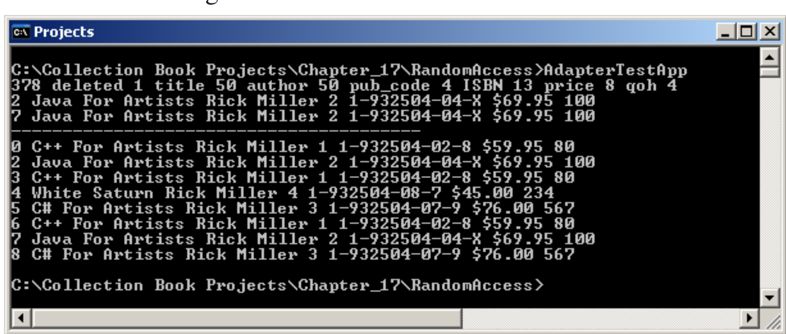


Figure 17-12: Results of Running Example 17.16 Once

Quick Review

You can conduct random access file I/O with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the file pointer at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

Working With Log Files

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

The following three examples together implement a simple logging system. It consists of three classes. The first, LogEntry, given in Example 17.17, represents the type of data that will be saved in the log file. The second, Logger, given in Example 17.18, implements the logging functionality with the help of several classes in the System.IO.Log namespace. The third class, LoggerTestApp, given in Example 17.19, tests the Logger class by writing several entries to the log and then reading the log and writing its contents to the console.

17.17 LogEntry.cs

```
using System;

[Serializable]
public class LogEntry {
  private string _subsystem;
  private int _severity;
  private string _text;
  private DateTime _timestamp;

  public DateTime TimeStamp {
    get { return _timestamp; }
    set { _timestamp = value; }
  }

  public string SubSystem {
    get { return _subsystem; }
    set { _subsystem = value; }
  }

  public int Severity {
    get { return _severity; }
    set { _severity = value; }
  }

  public string Text {
    get { return _text; }
    set { _text = value; }
  }

  public LogEntry(DateTime timestamp, string subsystem, int severity, string text){
    TimeStamp = timestamp;
    SubSystem = subsystem;
    Severity = severity;
    Text = text;
  }

  public override String ToString(){
    return TimeStamp.ToString() + " " + SubSystem + " " + Severity + " " + Text;
  }
} // end LogEntry class definition
```

Referring to Example 17-17 — the LogEntry class represents the data that will be captured and written to the log. A log entry will contain a TimeStamp property indicating when the event occurred, a SubSystem property indicating the subsystem of origin, Severity property indicating the severity of the event, and a Text property that contains the string with a detailed description of the event.

17.18 Logger.cs

```
using System;
using System.IO;
using System.IO.Log;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Serialization.Formatters.Binary;

public class Logger {
  private string _logfilename;
  private FileRecordSequence _sequence;
  private SequenceNumber _previous;


  public Logger(string logfilename){
```



```
15      _logfilename = logfilename;
16      _sequence = new FileRecordSequence(logfilename, FileAccess.ReadWrite);
17      _previous = SequenceNumber.Invalid;
18    }
19
20    public Logger():this("logfile.log"){ }
21
22    public void Append(LogEntry entry){
23      _previous = _sequence.Append(ToArraySegment(entry), SequenceNumber.Invalid,
24                          _previous, RecordAppendOptions.ForceFlush);
25    }
26
27    public ArraySegment<byte> ToArraySegment(LogEntry entry) {
28      MemoryStream stream = new MemoryStream();
29      BinaryFormatter formatter = new BinaryFormatter();
30      formatter.Serialize(stream, entry);
31      stream.Flush();
32      return new ArraySegment<byte>(stream.GetBuffer());
33    }
34
35    public String GetLogRecords() {
36      StringBuilder sb = new StringBuilder();
37      BinaryFormatter formatter = new BinaryFormatter();
38      IEnumerable<LogRecord> records = _sequence.ReadLogRecords(_sequence.BaseSequenceNumber,
39                                             LogRecordEnumeratorType.Next);
40      foreach (LogRecord record in records) {
41        LogEntry entry = (LogEntry) formatter.Deserialize(record.Data);
42        sb.Append(entry.ToString() + "\r\n");
43      }
44      return sb.ToString();
45    }
46
47    public void Dispose(){
48      _sequence.Dispose();
49    }
50  } // end class definition
```

Referring to the Example 17.18 — note that the Logger class uses a host of classes found in other namespaces. From the System.IO.Log namespace it uses the FileRecordSequence and SequenceNumber classes. The FileRecordSequence represents a sequence of log records stored in a simple file. SequenceNumbers are not numbers per se. They represent unique pointers from one log entry to the next within a sequence of log entries.

The Logger.Append() method on line 22 takes a LogEntry reference and in turn calls the FileRecordSequence.Append() method, which actually does the heavy lifting. The FileRecordSequence.Append() method has several overloaded variations. The one I use here requires that the log data being written be presented to it in the first argument as an array segment of bytes. *(i.e.*, ArraySegment<byte>) You'll find the ArraySegment generic structure in the System namespace. The Logger.ToArraySegment() method beginning on line 27 does the dirty work of converting a LogEntry object to a ArraySegment<byte> object.

The Logger.GetLogRecords() method on line 35 uses the FileRecordSequence.ReadLogRecords() method to read the records, converts them back into LogEntry objects, appends their string representation to a StringBuilder object, and ultimately returns the whole lot of them as one long string.

17.19 LoggerTestApp.cs

```
using System;
using System.Collections.Generic;
using System.Text;

 public class LoggeTestApp {
   static void Main(string[] args) {
     Logger logger = new Logger();
     LogEntry entry1 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
     LogEntry entry2 = new LogEntry(DateTime.Now, "Main Engine", 3, "Main condenser loss of vacuum");
     LogEntry entry3 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");
     LogEntry entry4 = new LogEntry(DateTime.Now, "Reactor", 1, "Loss of control rod control");
     LogEntry entry5 = new LogEntry(DateTime.Now, "Reactor Coolant", 3, "Main coolant pump speed limited");

     logger.Append(entry1);
     logger.Append(entry2);
     logger.Append(entry3);
     logger.Append(entry4);
     logger.Append(entry5);
     Console.Write(logger.GetLogRecords());
     logger.Dispose();
   } // end Main()
} // end class definition
```


Referring to Example 17.19 — the LoggerTestApp creates five LogEntry objects and calls the Logger.Append() method to insert each entry into the log. It then calls the Logger.GetLogRecords() and prints the results to the console.

To compile this program on Windows XP you'll need to do a couple of things. First, you'll need to have installed the .NET Framework 3.0 Redistributable. Second, locate the System.IO.Log.dll in the C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0 directory and add this path to your path environment variable. Once you set your path you'll need to compile the source files with the `/reference` switch to compile the files along with the System.IO.Log.dll like so:

```
csc /r:System.IO.Log.dll *.cs
```

Figure 17-13 shows the results of running the LoggerTestApp program one time. Running the program multiple times results in repeated log entries.

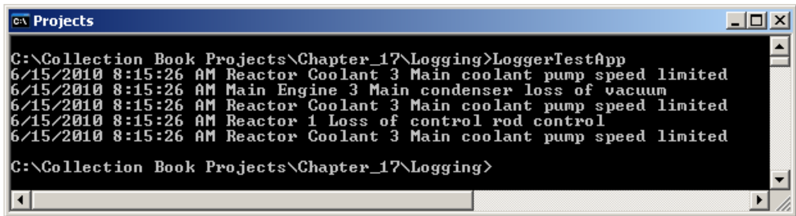


Figure 17-13: Results of Running Example 17.19

Quick Review

The System.IO.Log namespace contains classes, structures, interfaces, and enumerations designed to help you create robust event logging services for your programs. Some of the functionality provided by the contents of the System.IO.Log namespace is only available on Microsoft Windows 2003r2 and Windows Vista or later operating systems. These operating systems come with the Common Log File System (CLFS).

Using FileDialogs

As you know by now, the .NET Framework provides a large collection of GUI components that make programming rich graphical user interfaces relatively painless. Most of these classes can be found in the System.Windows.Forms namespace. Two of those classes: OpenFileDialog and SaveFileDialog make it easy to graphically select and open or save files. The following example uses the OpenFileDialog class to select one or more files to open and display several file properties in a TextBox. The example consists of two classes: GUI and MainApp.cs.

17.20 GUI.cs

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class GUI : Form {

  private SplitContainer _splitContainer1;
  private TextBox _textBox1;
  private Button _button1;

  public String TextBoxText {
    get {  return _textBox1.Text;  }
    set {  _textBox1.Text = value; }
  }

  public GUI(MainApp ma){
     this.InitializeComponent(ma);
  }

  private void InitializeComponent(MainApp ma) {
    _splitContainer1 = new SplitContainer();
    _textBox1 = new TextBox();
    _button1 = new Button();
    _splitContainer1.Panel1.SuspendLayout();
    _splitContainer1.Panel2.SuspendLayout();
```



```
    _splitContainer1.SuspendLayout();
    this.SuspendLayout();

    _splitContainer1.Dock = DockStyle.Fill;
    _splitContainer1.Location = new Point(0, 0);
    _splitContainer1.Panel1.Controls.Add(_textBox1);
    _splitContainer1.Panel2.Controls.Add(_button1);
    _splitContainer1.Size = new Size(292, 273);
    _splitContainer1.SplitterDistance = 161;
    _splitContainer1.TabIndex = 0;

    _textBox1.Location = new Point(3, 3);
    _textBox1.AutoSize = true;
    _textBox1.Anchor = (AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right);
    _textBox1.Multiline = true;
    _textBox1.Name = "textBox1";
    _textBox1.Size = new Size(155, 267);
    _textBox1.TabIndex = 0;

    _button1.Location = new Point(27, 12);
    _button1.Size = new System.Drawing.Size(75, 23);
    _button1.TabIndex = 0;
    _button1.Text = "Open File";
    _button1.UseVisualStyleBackColor = true;
    _button1.Click += new System.EventHandler(ma.Button1_Click);

    this.AutoScaleMode = AutoScaleMode.None;
    this.ClientSize = new System.Drawing.Size(292, 273);
    this.Controls.Add(_splitContainer1);

    this.Text = "FileDialog Demo";
    _splitContainer1.Panel1.ResumeLayout(false);
    _splitContainer1.Panel1.PerformLayout();
    _splitContainer1.Panel2.ResumeLayout(false);
    _splitContainer1.ResumeLayout(false);
    this.ResumeLayout(false);
  } // End InitializeComponent()
} // End class definition
```

Referring to Example 17.20 — the GUI class inherits from Form and uses a SplitContainer to hold a TextBox and a Button. The TextBox.MultiLine property is set to true and its Anchor property is set to anchor to all four sides of its containing panel. The button's Click event is set to invoke the MainApp.Button1_Click() method.

17.21 MainApp.cs

```
using System;
using System.Windows.Forms;
using System.Text;
using System.IO;

public class MainApp {
  private OpenFileDialog _fileDialog;
  private GUI _gui;

  public MainApp(){
    _gui = new GUI(this);
    _fileDialog = new OpenFileDialog();
    _fileDialog.Multiselect = true;
    Application.Run(_gui);
  }

  public void Button1_Click(Object o, EventArgs e){
    _fileDialog.ShowDialog();
    String[] filenames = _fileDialog.FileNames;
    StringBuilder sb = new StringBuilder();
    foreach(String s in filenames){
      FileInfo file = new FileInfo(s);
      sb.Append("FileName:" +  file.Name + "\r\n");
      sb.Append("Directory:" + file.DirectoryName + "\r\n");
      sb.Append("Size:" + file.Length + " Bytes\r\n");
      sb.Append("\r\n");
    }
    _gui.TextBoxText = sb.ToString();
  }

  public static void Main(){
    new MainApp();
  }
} // end class definition
```

Referring to Example 17.21 — the MainApp class plays host to the Main() method and the Button1_Click() event handler method. In the body of the MainApp constructor the OpenFileDialog object is created and it's Multiselect property is set to true. This allows the user to select multiple files to open at the same time.

When the button is clicked in the GUI, the Button1_Click() event handler method calls the OpenFileDialog's ShowDialog() method. This displays the dialog and lets users select the file(s) they wish to open. At this point the program effectively blocks until the user clicks the Open button on the OpenFileDialog window.

The OpenFileDialog.FileNames property returns a string array containing the names of the file(s) selected by the user. The `foreach` statement starting on line 21 iterates over each filename, creates a FileInfo object, extracts the required information about each file, and appends it to a StringBuilder object. When the `foreach` statement finishes, the file information contained in the StringBuilder object is written to the GUI.TextBoxText property, which in turn sets its TextBox's Text property.

Figure 17-14 shows the results of running this program and selecting three files named GUI.cs, MainApp.cs, and MainApp.exe. Your results will differ depending on what files you select.

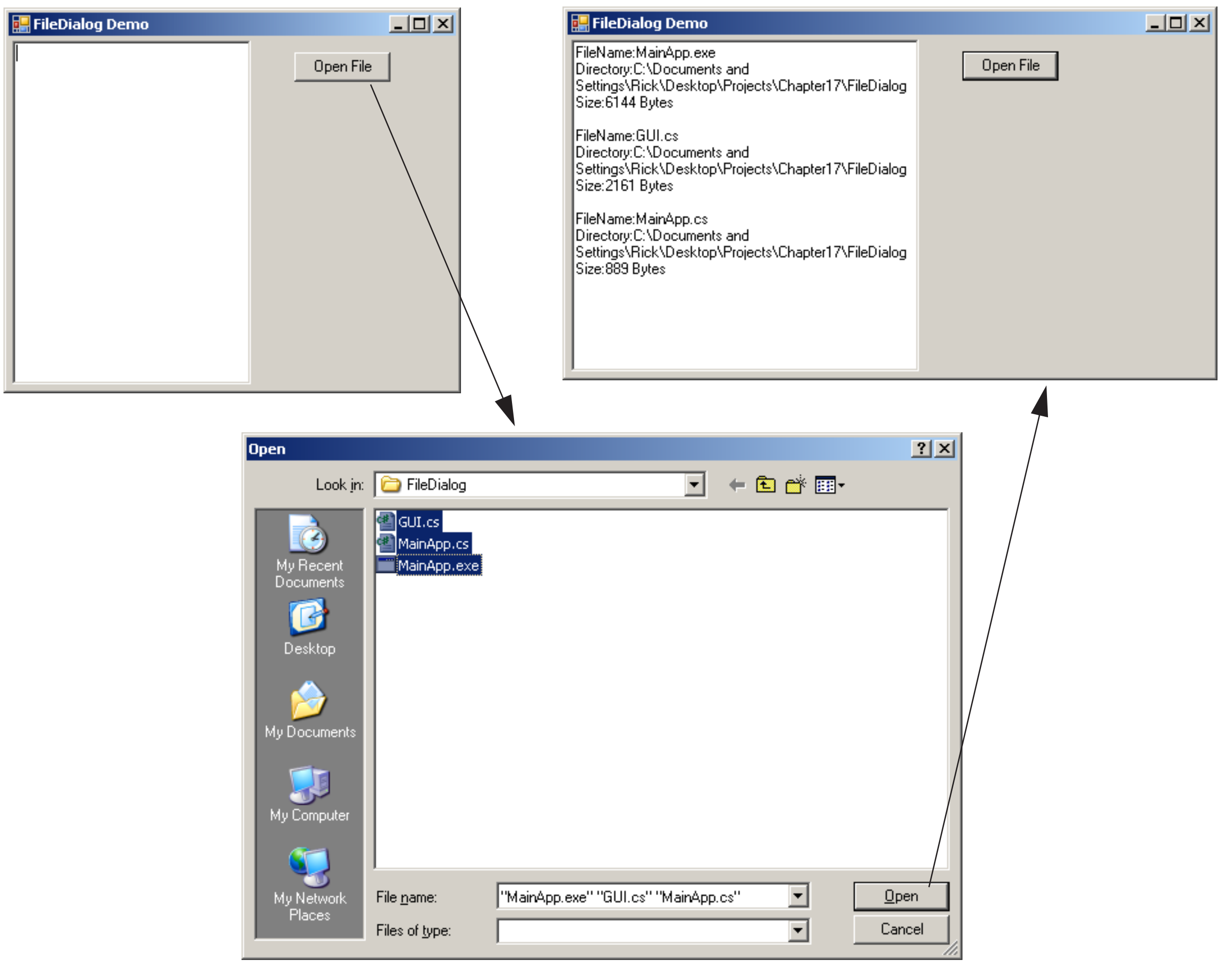


Figure 17-14: Results of Running Example 17.21 and Selecting Three Files

Quick Review

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner, the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

Persisting a BindingList<T> Collection

I'd like to revisit the BindingListDataGridDemo project presented in Chapter 16 and show you how to automatically persist the SortableBindingList<T> collection each time a change is made to the data in the DataGridView control.

As you've learned from reading the section on serialization, all the objects within a collection must be serializable so that you can serialize the collection and its items to disk. This means that the Person class, the PersonKey class, and the SortableBindingList<T> class must be tagged with the [Serializable] attribute. Examples 17.22 through 17.24 gives the modified source files for these classes.

17.22 Person.cs (Tagged with [Serializable] attribute)

```
using System;
using System.ComponentModel;

[Serializable]
public class Person : IComparable, IComparable<Person>, INotifyPropertyChanged {

  //enumeration
  public enum Sex {MALE, FEMALE};

  //event
  public event PropertyChangedEventHandler PropertyChanged;

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;

  public Person(){
    _firstName = string.Empty;
    _middleName = string.Empty;
    _lastName = string.Empty;
    _gender = Person.Sex.MALE;
    _birthday = DateTime.Now;
    _dna = Guid.NewGuid();
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
    get { return _firstName; }
    set { _firstName = value;
          NotifyPropertyChanged("FirstName");
        }
```



```
    }

    public String MiddleName {
      get {  return _middleName; }
      set {  _middleName = value;
            NotifyPropertyChanged("MiddleName");
          }
    }

    public String LastName {
      get {  return _lastName; }
      set {  _lastName = value;
            NotifyPropertyChanged("LastName");
          }
    }

    public Sex Gender {
      get {  return _gender; }
      set {  _gender = value;
            NotifyPropertyChanged("Gender");
          }
    }

    public DateTime Birthday {
      get {  return _birthday; }
      set {  _birthday = value;
            NotifyPropertyChanged("Birthday");
          }
    }

    public Guid DNA {
      get {  return _dna; }
      set {  _dna = value;
            NotifyPropertyChanged("DNA");
      }
    }

    public int Age {
       get {
        int years = DateTime.Now.Year - _birthday.Year;
         int adjustment = 0;
        if(DateTime.Now.Month < _birthday.Month){
           adjustment = 1;
         }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
             adjustment = 1;
         }
      return years - adjustment;
      }
     }

    public String FullName {
      get {  return FirstName + " " + MiddleName + " " + LastName; }
    }

    public String FullNameAndAge {
      get {  return FullName + " " + Age; }
    }

    protected String SortableName {
      get {  return LastName + FirstName + MiddleName; }
    }

    public PersonKey Key {
      get {  return new PersonKey(this.ToString()); }
    }

    public override String ToString(){
      return (FullName + "  " + Gender + "  " + Age + " " + DNA);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(Person) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public static bool operator ==(Person lhs, Person rhs){
```



```
146     return lhs.Equals(rhs);
147   }
148
149   public static bool operator !=(Person lhs, Person rhs){
150     return !(lhs.Equals(rhs));
151   }
152
153   public int CompareTo(object obj){
154     if((obj == null) || (typeof(Person) != obj.GetType()))  {
155       throw new ArgumentException("Object is not a Person!");
156     }
157     return this.SortableName.CompareTo(((Person)obj).SortableName);
158   }
159
160   public int CompareTo(Person p){
161     if(p == null){
162       throw new ArgumentException("Cannot compare null objects!");
163     }
164     return this.SortableName.CompareTo(p.SortableName);
165   }
166
167
168   private void NotifyPropertyChanged(string propertyName){
169     if(PropertyChanged != null){
170       PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
171     }
172   }
173 } // end Person class
```

Referring to example 17.22 — I added the [Serializable] attribute on line 4 just above the start of the class definition.

17.23 PersonKey.cs (Tagged with [Serializable] attribute)

```
using System;

[Serializable]
public class PersonKey : IEquatable<String>, IComparable, IComparable<PersonKey> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public int CompareTo(object obj){
     return _keyString.CompareTo(obj);
    }


    public int CompareTo(PersonKey pk){
      return _keyString.CompareTo(pk._keyString);
    }
}
```

Referring to example 17.23 — again, I added the [Serializable] attribute on the line above the start of the class definition.

17.24 SortableBindingList<T> (Tagged with [Serializable] attribute)

```
using System;
using System.ComponentModel;
using System.Collections.Generic;

[Serializable]
public class SortableBindingList<T> : BindingList<T> {
```



```
7
8     public void Sort(){
9       ((List<T>)Items).Sort();
10    }
11  }
```

Referring to example 17.24 — the [Serializable] attribute appears on line 5. OK, now that the preliminaries are out of the way, it's time to modify the original example to add the persistence capability. There are many ways to integrate this example with a persistence mechanism. I chose to serialize the entire collection to disk every time a change is made to the collection. I do this by taking advantage of the events that fire in response to collection changes. Let's take a look at the code and I'll explain what I did.

17.25 PersistedBindingListDemo.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;
using System.Data;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class PersistedBindingListDemo : Form {

 #region Fields
  SortableBindingList<Person> _personList;
  DataGridView _dataGridView;
  TableLayoutPanel _mainPanel;
  TableLayoutPanel _buttonPanel;
  Button _button1;
  Button _button2;
 #endregion

 #region Constructor
   public PersistedBindingListDemo(){
     InitializeBindingList();
     InitializeComponent();
   }
 #endregion

 #region InitializationMethods
   private void InitializeBindingList(){
      FileStream fs = null;
      try{
        FileInfo personFile = new FileInfo("person.dat");
        if(personFile.Exists){
          fs = new FileStream("person.dat", FileMode.Open);
          BinaryFormatter bf = new BinaryFormatter();
          _personList = (SortableBindingList<Person>)bf.Deserialize(fs);
        }else{
          _personList = new SortableBindingList<Person>();
        }
      }catch(Exception e){
        MessageBox.Show("Problem deserializing Person data file. Full error -> " + e);
      }
      finally{
        if(fs != null){
          fs.Close();
        }
      }
      _personList.AddingNew += AddingNew_Handler;
      _personList.ListChanged += ListChanged_Handler;
      _personList.AllowNew = true;
      _personList.AllowEdit = true;
      _personList.AllowRemove = true;
      _personList.RaiseListChangedEvents = true;
   }

   private void InitializeComponent(){
     _mainPanel = new TableLayoutPanel();
     _mainPanel.RowCount = 2;
     _mainPanel.ColumnCount = 1;
     _mainPanel.Dock = DockStyle.Fill;

     _buttonPanel = new TableLayoutPanel();
     _buttonPanel.RowCount = 1;
     _buttonPanel.ColumnCount = 2;
     _buttonPanel.Dock = DockStyle.Fill;

```



```
    InitializeDataGridView();

    _button1 = new Button();
    _button1.Text = "Sort";
    _button1.Click += SortButton_Handler;

    _button2 = new Button();
    _button2.Text = "Delete";
    _button2.Click += DeleteButton_Handler;

    _buttonPanel.Controls.Add(_button1);
    _buttonPanel.Controls.Add(_button2);

    _mainPanel.Controls.Add(_dataGridView);
    _mainPanel.Controls.Add(_buttonPanel);

    this.Controls.Add(_mainPanel);
    this.Width = 850;
    this.Height = 250;
    this.Text = "BindingListDataGridDemo";
  }


  private void InitializeDataGridView(){
    _dataGridView = new DataGridView();
    _dataGridView.Dock = DockStyle.Fill;
    DataGridViewComboBoxColumn genderColumn = new DataGridViewComboBoxColumn();
    genderColumn.DataSource = Enum.GetValues(typeof(Person.Sex));
    genderColumn.DataPropertyName = "Gender";
    genderColumn.HeaderText = "Gender";
    _dataGridView.Columns.Add(genderColumn);
    _dataGridView.DataSource = _personList;
    _dataGridView.EditMode = DataGridViewEditMode.EditOnEnter;
    _dataGridView.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    _dataGridView.DataBindingComplete += DataBindingComplete_Handler;
  }
#endregion


#region EventHandlerMethods

  public void AddingNew_Handler(object sender, AddingNewEventArgs e){
     e.NewObject = new Person();
     Console.WriteLine("New Person object created!");
  }

  public void ListChanged_Handler(object sender, ListChangedEventArgs e){
    switch(e.ListChangedType){
      case ListChangedType.ItemDeleted:
              Console.WriteLine("Item successfully deleted.");
              foreach(Person p in _personList){
                Console.WriteLine(p);
              }
              SerializeBindingList(_personList);
              break;
      case ListChangedType.ItemChanged:
              ((CurrencyManager)_dataGridView.BindingContext[_personList]).Refresh();
              Console.Write("Item successfully updated. Property: " + e.PropertyDescriptor.Name );
              Console.WriteLine(" - Value: " + e.PropertyDescriptor.GetValue(_personList[e.NewIndex]));
              SerializeBindingList(_personList);
              break;
    }
  }


 public void SortButton_Handler(object sender, EventArgs e){
   _personList.Sort();
   ((CurrencyManager)_dataGridView.BindingContext[_personList]).Refresh();
   SerializeBindingList(_personList);
 }


 public void DeleteButton_Handler(object sender, EventArgs e){
     if(_personList.Count > 0){
       _personList.RemoveAt(_dataGridView.CurrentRow.Index);
       Console.WriteLine("Person object deleted!");
     }
 }

 public void DataBindingComplete_Handler(object sender, EventArgs e){
   _dataGridView.Columns["FullNameAndAge"].Visible = false;
```



```
149      _dataGridView.Columns["FullName"].Visible = false;
150      _dataGridView.Columns["Key"].Visible = false;
151      _dataGridView.Columns["DNA"].ReadOnly = true;
152      _dataGridView.Columns["DNA"].ToolTipText = "Read Only!";
153      _dataGridView.Columns["Birthday"].ToolTipText = "Format: mm/dd/yyyy";
154
155      for(int i=0; i<_dataGridView.Columns.Count; i++){
156        _dataGridView.Columns[i].Width = 100;
157      }
158
159      _dataGridView.Columns["DNA"].Width = 225;
160      _dataGridView.Columns["Age"].Width = 50;
161      _dataGridView.Columns["FirstName"].DisplayIndex = 0;
162      _dataGridView.Columns["MiddleName"].DisplayIndex = 1;
163      _dataGridView.Columns["LastName"].DisplayIndex = 2;
164    }
165
166   #endregion
167
168
169   #region UtilityMethods
170
171     private void SerializeBindingList(SortableBindingList<Person> _personList){
172       FileStream fs = null;
173       try{
174            fs = new FileStream("person.dat", FileMode.Create);
175            BinaryFormatter bf = new BinaryFormatter();
176            bf.Serialize(fs, _personList);
177        }catch(Exception e){
178          MessageBox.Show("Problem serializing Person list to data file. Full error -> " + e);
179        }
180        finally{
181          if(fs != null){
182            fs.Close();
183          }
184       }
185     }
186
187   #endregion
188
189   #region MainMethod
190     [STAThread]
191     public static void Main(){
192       Application.Run(new PersistedBindingListDemo());
193     }
194   #endregion
195 }
```

Referring to example 17.25 — you can compare this example with the original presented in chapter 16, example 16.7. First off, I added three new using directives to gain shorthand access to the System.IO, System.Runtime.Serialization, and System.Runtime.Serialization.Formatters.Binary namespaces.

Next, I modified the InitializeBindingList() method on line 30. The first thing I do is declare a FileStream reference named `fs` and initialize it to null. I do this outside of the `try/catch` block because I need access to the `fs` reference in the `finally` clause. In the body of the `try` block I declare and initialize a FileInfo reference named personFile. This is tied to a file in the working directory named “person.dat”. I then check to see if the person.dat file exists. If so, I create the FileStream object and a BinaryFormatter and deserialize the person.dat file, assigning the result to the _personList reference. If the person.dat file does not exist, I simply initialize the _personList reference with a new SortableBindingList<Person> object. The rest of the InitializeBindingList() method proceeds like the original version.

Now, the next big change comes in the form of a utility method named SerializeBindingList() which begins on line 171. It takes a SortableBindingList<Person> as an argument and serializes the list to the person.dat file.

I then sprinkle the SerializeBindingList() method around the application where it’s needed. I added it to the ListChanged_Handler() method on lines 121 and 127. Thus whenever an item is deleted or changed the list is written to disk. I also added the SerializeBindingList() method to the SortButton_Handler() method.

The net result of these changes is that when you launch the application it will try and load the person.dat file from the working directory if the file exists, and every time you add, edit, or delete a person or sort the collection, the entire collection will be automatically serialized to disk. Figure 17-15 shows the results of running this program.

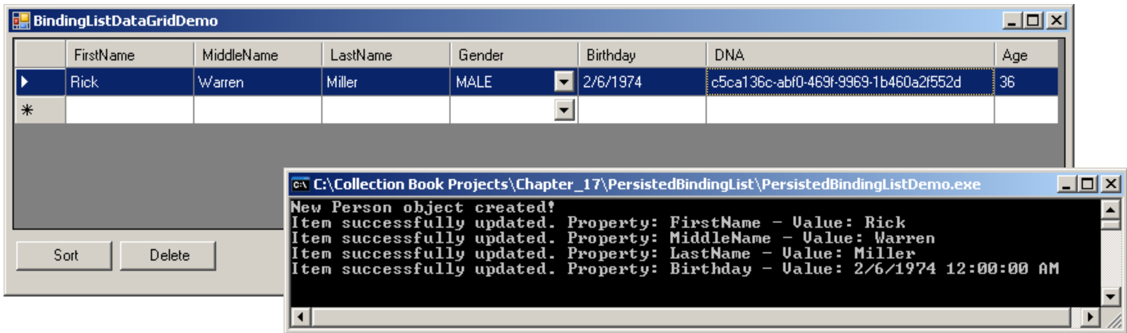


Figure 17-15: Results of Running Example 17.25

Summary

In most all cases, data generated by an application and stored on an auxiliary storage device such as a hard disk, is saved as an organized, related collection of bits in a structure commonly referred to as a *file*.

It is the operating system's responsibility to manage the organization, reading, and writing of files. When a new storage device is added to your computer, it must first be formatted in a way that allows the operating system to access its data.

The file, from the operating system's point of view, is the fundamental storage organizational element. An application's associated data can be stored in one or more files. A file is located in another organizational element called a *directory*. A directory is a special type of file that contains a list of files and directories. A directory contained inside another directory is called a *subdirectory*.

The topmost directory structure is referred to as the *root* directory. The root directory of a particular drive is indicated by the name of the drive followed by a colon ':', followed by a backward slash character '\'. The root directory of the C drive would be "C:\".

The location of a particular file within a directory structure is indicated by a string of characters called a *path*. The path to the file's location can be *absolute* or *relative*. An *absolute path* includes the name or letter of the drive and all directory and subdirectory names required to pinpoint the file's location. A *relative path* is the path to a file from some arbitrary starting point, usually a working directory.

You can easily create and manipulate directories and files with the help of several classes provided in the .NET Framework System.IO namespace. These include the *Path*, *File*, *FileInfo*, *Directory*, *DirectoryInfo*, and *DriveInfo* classes.

Verbatim strings are formulated by preceding the string with the '@' character which signals the compiler to interpret the string literally, including special characters and line breaks.

Object serialization provides an easy, convenient way for you to persist application data to disk. Object serialization is also the least flexible way to store application data because you can't edit the resulting file. Use a FileStream object and a BinaryFormatter to serialize objects to disk. Before an object can be serialized it must be tagged as being serializable with the serializable attribute. Place the serializable attribute above the class declaration line.

When *serializing* a collection of objects, remember that all objects contained within the collection must be serializable. You don't have to worry about the collections themselves, including ordinary arrays, as they are already tagged as being serializable.

You can get around the limitation of ordinary serialization by serializing objects to disk in XML format. Use the StreamWriter and XMLSerializer classes to serialize objects to disk in XML format. Use a FileStream and XMLSerializer to deserialize objects from an XML file.

The StreamReader and StreamWriter classes let you read and write text files. Text files are usually processed line-by-line. Lines of text are terminated with the special characters *carriage-return and line-feed* (\r\n). Each line can contain one or more fields *delimited* by some character. The comma ',' is a commonly used field delimiter. Individual fields can be further delimited as required.

Look to the objects in your program to determine the type of information your text file(s) must contain. You'll need to save enough data to recreate objects.

Process a text file by reading each line and breaking it into *tokens* with the String.Split() method. If one or more fields are also delimited, use the String.Split() method to tokenize the data as required.

Use the BinaryReader and BinaryWriter classes to read and write binary data to disk. The BinaryWriter class provides an overloaded Write() method that is used to write each of the simple types including strings and arrays of bytes and characters. The BinaryReader class provides an assortment of Read*Typename*() methods where *Typename* may be any one of the simple types to include strings and arrays of bytes and characters.

You can conduct *random access file I/O* with the BinaryReader, BinaryWriter, and FileStream classes. The FileStream class provides a Seek() method that allows you to position the *file pointer* at any point within a file. As you learned in the previous section, the BinaryReader and BinaryWriter classes provide methods for reading and writing binary, string, byte, and character array data.

Use the OpenFileDialog and SaveFileDialog classes to graphically select and open/save files. The OpenFileDialog can be used to select multiple files simultaneously. When used in this manner the OpenFileDialog.FileNames property returns a string array containing the names of the files selected.

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes

Chapter 18

Nikon F3HP
American Flag Truck

Creating Custom Collections

Learning Objectives

- State the benefits of extending an existing collection vs. creating one from scratch
- Explain why you'd extend an existing collection vs. implement an interface and vice versa
- Understand the difference between the IEnumerable and IEnumerable<T> interfaces
- Implement the IEnumerable and IEnumerable<T> interfaces in a custom collection
- Understand the purpose and use of iterator blocks
- Implement iterator blocks in a custom collection
- State the purpose of the yield keyword when used in an iterator block
- Implement named iterators to provide alternative collection ordering
- State why it would be necessary to implement custom serialization
- Implement XML serialization and deserialization in a custom collection
- Create event enabled collections
- Implement the INotifyCollectionChanged interface

Introduction

The .NET framework offers a wide assortment of collection classes that will satisfy most of your programming needs. However, you will eventually encounter a situation that dictates a custom solution. This chapter will show you how to create a custom collection class either from scratch or by extending an existing collection.

I'll start with a brief discussion to help you decide when and if a custom collection class is an appropriate solution. As a rule, it's best to stick with the collection classes provided by the .NET framework. They've been thoroughly tested by programmers like you all over the planet. Also, the introduction of generic collections eliminated the need to create one-up, strongly-typed collections.

Next I'll show you how to create a custom collection by extending an existing collection. The .NET framework provides several collection types specifically intended for use as base classes for custom collections. You saw an example of this in chapter 16 when I extended the BindingList<T> class to create a SortableBindingList<T> class.

If you must create a custom collection you will need to make a decision regarding to what degree your class will adopt expected collection behavior. For example, if you want your custom collection to be iterated by the `foreach` statement you'll need to implement the IEnumerable and IEnumerable<T> interfaces. I'll also show you how to implement and use named iterators so you can provide alternative enumerations.

If your custom collection class is particularly special you may need to implement custom serialization. Also, if you want to respond to changes to your collection's items or to one of its properties, you'll need to implement the INotifyCollectionChanged and INotifyPropertyChanged interfaces, or, if the situation dictates, you can create your own custom events.

The topics covered in this chapter bring together a lot of material presented earlier and separately throughout the book. Some of it's a review; some is new material.

When you've finished reading this chapter you will be confident in your abilities to create a custom collection when the need arises.

Deciding When To Create A Custom Collection Class

The first question you must answer when you start thinking about creating a custom collection is: "Do I really need to create a custom collection or will one of the existing collections satisfy my needs?" This question has multiple answers. First, an existing collection may completely fill the bill. Perhaps all you need to do is to study the documentation to see how it suits your needs. The primary benefit gained from using an existing collection is that it saves you a ton of time. Secondly, you can extend an existing collection and with a small amount of extra code create a custom collection that satisfies your needs while saving you the work of writing one from scratch. If none of these scenarios work for you then creating a custom collection from scratch looks like your only alternative. The following list of considerations will help you decide if creating a custom collection is the right development path to pursue:

- None of the existing collection classes meet your needs
- You have a better way of doing something, like an improved algorithm
- You want to provide an alternative enumeration (i.e., a named iterator)
- You want to provide a custom iterator to perform special processing
- You need to implement custom serialization to handle complex or evolving data structures

Extending An Existing Collection

Extending an existing collection class allows you to create a custom collection while leveraging the hard work done by the .NET framework engineers. In this section I'll show you how to extend several non-generic and generic collection classes to create custom collections. Some of the techniques I'll demonstrate, especially the extension of a non-generic collection to hide casting operations, has been superseded by the introduction of generic collection classes. However, you may find yourself maintaining legacy code. Believe it or not, there exist production environ-

ments that cannot be automatically updated to the latest and greatest version of the .NET framework due to security or connectivity reasons.

Extending Non-Generic ArrayList

The original, non-generic collection classes are implemented in terms of System.Object. By this I mean you can store any type of object in a non-generic collection but when you access an element in the collection it is returned to you as a System.Object. Say for example you insert Person objects into an ArrayList. To use the interface methods available to Person objects when retrieved from the list you'll need to cast the retrieved object to type Person.

You can create what is referred to as a strongly-typed ArrayList by extending the ArrayList class and overriding or implementing new methods in the subclass that perform type checking upon insertions into the list and automatic casting to the appropriate type on access. (**Note:** You can extend practically any non-generic collection in the same manner, not just ArrayList.)

Example 18.1 demonstrates the extension of the ArrayList class to create a strongly-typed list of integers.

18.1 IntArrayList.cs

```
using System;
using System.Collections;

public class IntArrayList : ArrayList {

    public override int Add(object value){
      if(value.GetType() != typeof(System.Int32)){
        throw new ArgumentException("Incoming object is not an Int32!");
      }
      return base.Add(value);
    }

    public new int this[int i]{
      get { return (int) base[i]; }
      set { base[i] = (int)value; }
    }
}
```

Referring to example 18.1 — in this example I've overridden the Add() method and added type-checking upon insertion into the list. If the incoming object fails the type-check I throw an ArgumentException. I've also implemented a new indexer. I couldn't override the indexer in this case because the overridden indexer would still return an object, which wouldn't do much good if I wanted integers. Example 18.2 demonstrates the use of the IntArrayList in a short program.

18.2 MainApp.cs

```
1   using System;
2   using System.Collections;
3
4   public class MainApp {
5     public static void Main(){
6       IntArrayList list = new IntArrayList();
7
8       for(int i=0; i<25; i++){
9         list.Add(i);
10      }
11
12      for(int i=0; i<list.Count; i++){
13        Console.Write(list[i] + " ");
14      }
15
16      Console.WriteLine();
17
18      try{
19        list.Add("Hello World!");
20      }catch(ArgumentException ae){
21        Console.WriteLine(ae);
22      }
23    }
24  }
```

Referring to example 18.2 — I create an instance of IntArrayList on line 6 and in the body of the `for` loop on line 8 I insert 25 integer values. In the `for` loop on line 12 I print the values in the list to the console. In the `try` block beginning on line 18 I try adding a string to the list. This throws an ArgumentException which is handled in the `catch` clause. Figure 18-1 shows the results of running this program.

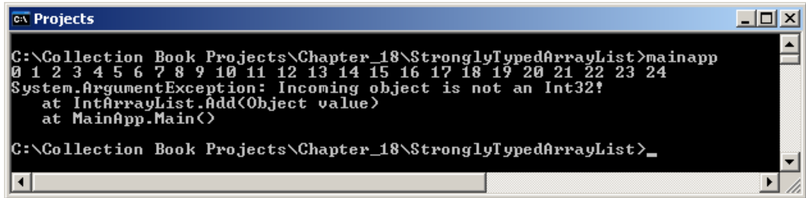


Figure 18-1: Results of Running Example 18.2

Superseded By Generics

Note that the need to extend a non-generic class to create a strongly-typed collection has been eliminated with the introduction of generic collections. To create a list of integers simply declare a List<int>. As you learned in chapter 5, generic collections perform better than their non-generic counterparts, especially when dealing with value types because they avoid the performance penalties associated with boxing and unboxing.

Gaining More Control Over The Custom Collection

The problem with extending ArrayList is that its public methods are still public and therefore still accessible. From a purely polymorphic perspective, it's not a pretty situation. For example, while declaring a reference to an ArrayList but having the reference actually point to an IntArrayList works well for the overridden Add() method, the indexer gives us problems because it's declared as "new" in the subclass. The "new" implementation of the indexer will only be called if the reference is declared to be an IntArrayList. It's these types of unpleasantries that make extending specific non-generic classes awkward. Fortunately, the .NET framework provides collection classes more suitable for the creation of custom collections.

For non-generic custom collections the preferred base class is CollectionBase. If you compare CollectionBase with ArrayList you'll see that the CollectionBase class contains many more protected methods than does the ArrayList. The protected methods are accessible to subclasses (vertically) but not accessible horizontally. For example, the CollectionBase class does not provide a public indexer (identified as the Item property). Instead, it leaves the implementation of a strongly-typed indexer up to you.

For generic collections the Collection<T> class provides the same service as the non-generic CollectionBase class.

Examples 18.3 through 18.6 demonstrate the extension of the non-generic CollectionBase class to create a custom, strongly-typed PersonCollection class which holds objects of type Person. Example 18.3 gives the code for the Person class used in this example.

18.3 Person.cs

```
using System;
using System.ComponentModel;

public class Person : IComparable, IComparable<Person>, INotifyPropertyChanged {

  //enumeration
  public enum Sex {MALE, FEMALE};

  //event
  public event PropertyChangedEventHandler PropertyChanged;

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;

  public Person(){
    _firstName = string.Empty;
    _middleName = string.Empty;
    _lastName = string.Empty;
    _gender = Person.Sex.MALE;
    _birthday = DateTime.Now;
```



```
  _dna = Guid.NewGuid();

}

public Person(String firstName, String middleName, String lastName,
              Sex gender, DateTime birthday, Guid dna){
   FirstName = firstName;
   MiddleName = middleName;
   LastName = lastName;
   Gender = gender;
   Birthday = birthday;
   DNA = dna;
}

public Person(String firstName, String middleName, String lastName,
              Sex gender, DateTime birthday){
   FirstName = firstName;
   MiddleName = middleName;
   LastName = lastName;
   Gender = gender;
   Birthday = birthday;
   DNA = Guid.NewGuid();
}

public Person(Person p){
   FirstName = p.FirstName;
   MiddleName = p.MiddleName;
   LastName = p.LastName;
   Gender = p.Gender;
   Birthday = p.Birthday;
   DNA = p.DNA;
}

// public properties
public String FirstName {
  get {  return _firstName; }
  set {  _firstName = value;
        NotifyPropertyChanged("FirstName");
      }
}

public String MiddleName {
  get {  return _middleName; }
  set {  _middleName = value;
        NotifyPropertyChanged("MiddleName");
      }
}

public String LastName {
  get {  return _lastName; }
  set {  _lastName = value;
        NotifyPropertyChanged("LastName");
      }
}

public Sex Gender {
  get {  return _gender; }
  set {  _gender = value;
        NotifyPropertyChanged("Gender");
      }
}

public DateTime Birthday {
  get {  return _birthday; }
  set {  _birthday = value;
        NotifyPropertyChanged("Birthday");
      }
}

public Guid DNA {
  get {  return _dna; }
  set {  _dna = value;
        NotifyPropertyChanged("DNA");
  }
}

public int Age {
   get {
    int years = DateTime.Now.Year - _birthday.Year;
     int adjustment = 0;
    if(DateTime.Now.Month < _birthday.Month){
```

```
        adjustment = 1;
      }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
          adjustment = 1;
      }
    return years - adjustment;
    }
   }

  public String FullName {
    get { return FirstName + " " + MiddleName + " " + LastName; }
  }

  public String FullNameAndAge {
    get { return FullName + " " + Age; }
  }

  protected String SortableName {
    get { return LastName + FirstName + MiddleName; }
  }

  public PersonKey Key {
    get { return new PersonKey(this.ToString()); }
  }

  public override String ToString(){
    return (FullName + "  " + Gender + "  " + Age + " " + DNA);
  }

  public override bool Equals(object o){
    if(o == null) return false;
    if(typeof(Person) != o.GetType()) return false;
    return this.ToString().Equals(o.ToString());
  }

  public override int GetHashCode(){
    return this.ToString().GetHashCode();
  }

  public static bool operator ==(Person lhs, Person rhs){
    return lhs.Equals(rhs);
  }

  public static bool operator !=(Person lhs, Person rhs){
    return !(lhs.Equals(rhs));
  }

  public int CompareTo(object obj){
    if((obj == null) || (typeof(Person) != obj.GetType()))  {
      throw new ArgumentException("Object is not a Person!");
    }
    return this.SortableName.CompareTo(((Person)obj).SortableName);
  }

  public int CompareTo(Person p){
    if(p == null){
      throw new ArgumentException("Cannot compare null objects!");
    }
    return this.SortableName.CompareTo(p.SortableName);
  }


  private void NotifyPropertyChanged(string propertyName){
    if(PropertyChanged != null){
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
  }
} // end Person class
```

Referring to example 18.3 — the Person class implements the IComparable, IComparable<Person>, and INotifyPropertyChanged interfaces. Example 18.4 gives the code for the PersonKey class.

18.4 PersonKey.cs

```
using System;

public class PersonKey : IEquatable<String>, IComparable, IComparable<PersonKey> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }
```



```

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public int CompareTo(object obj){
     return _keyString.CompareTo(obj);
    }


    public int CompareTo(PersonKey pk){
      return _keyString.CompareTo(pk._keyString);
    }
}
```

Referring to example 18.4 — the PersonKey is a custom key class used in the Person class, so it’s required to include it to compile the code. Example 18.5 gets to the heart of the matter and gives the code for the custom PersonCollection class.

18.5 PersonCollection.cs

```
using System;
using System.Collections;

public class PersonCollection : CollectionBase {

  public Person this[int index] {
    get { return (Person) List[index]; }
    set { List[index] = value; }
  }

  public int Add(Person value){
    return List.Add(value); // List.Add() adds value to end of list and returns its index
  }

  public int IndexOf(Person value){
    return List.IndexOf(value);
  }

  public void Insert(int index, Person value){
    List.Insert(index, value); // May throw ArgumentOutOfRange exception
                               // if index falls outside array boundaries
  }

  public void Remove(Person value){
    List.Remove(value); //May throw ArgumentException if value not found
  }

  public bool Contains(Person value){
    return List.Contains(value);
  }

  protected override void OnInsert(int index, object value){
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("object inserted at index {0} is {1}", index, value);
    Console.ForegroundColor = ConsoleColor.Gray;
  }

  protected override void OnRemove(int index, object value){
    Console.ForegroundColor = ConsoleColor.Blue;
    Console.WriteLine("object removed from index {0} is {1}", index, value);
    Console.ForegroundColor = ConsoleColor.Gray;
  }

  protected override void OnSet(int index, object oldValue, object newValue){
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("object set at index {0} was {1} but is now {2}", index, oldValue, newValue);
    Console.ForegroundColor = ConsoleColor.Gray;
```



```
  }

  protected override void OnValidate(object value){
    if((value == null) || (value.GetType() != typeof(Person))){
      throw new ArgumentException("value must be of type Person and cannot be null!");
    }
  }
} // end class definition
```

Referring to example 18.5 — the PersonCollection extends the non-generic CollectionBase class and provides implementations for the indexer and the Add(), IndexOf(), Insert(), Remove(), and Contains() methods. To have some fun I've also provided implementations for the OnInsert(), OnRemove(), OnSet(), and OnValidate() methods. These methods are fired in response to their corresponding collection manipulation events. For example, the OnInsert() method is called when an object is inserted into the collection with the Insert() method.

Example 18.6 demonstrates the use of the PersonCollection class in a short program.

18.6 MainApp.cs (Demonstrating PersonCollection)

```
using System;
using System.Collections;

public class MainApp {
  public static void Main(){
    PersonCollection pc = new PersonCollection();
    Person p1 = new Person("Deekster", "Willis", "Jaybones", Person.Sex.MALE, new DateTime(1966, 02, 19));
    Person p2 = new Person("Knut", "J", "Hampson", Person.Sex.MALE, new DateTime(1972, 04, 23));
    Person p3 = new Person("Katrina", "Kataline", "Kobashar", Person.Sex.FEMALE, new DateTime(1982, 09, 3));
    Person p4 = new Person("Dreya", "Babe", "Weber", Person.Sex.FEMALE, new DateTime(1978, 11, 25));
    Person p5 = new Person("Sam", "\"The Guitar Man\"", "Miller", Person.Sex.MALE,
                           new DateTime(1988, 04, 16));
    Console.WriteLine("-------- Test the Add() method ---------------------------");
    pc.Add(p1);
    pc.Add(p2);
    pc.Add(p3);

    foreach(Person p in pc){
      Console.WriteLine(p.FullName);
    }

    Console.WriteLine("----- Remove: {0} ----------------------", p1.FullName);
    pc.Remove(p1);

    foreach(Person p in pc){
      Console.WriteLine(p.FullName);
    }

    Console.WriteLine("---------- Add another person ---------------");
    pc.Add(p4);

    foreach(Person p in pc){
      Console.WriteLine(p.FullName);
    }

    Console.WriteLine("--------- Test the indexer read -------------------------------");
    Console.WriteLine(pc[1].FullName);
    Console.WriteLine("--------- Test the indexer write -------------------------------");

    pc[0] = new Person("Slate", "Bo", "Hopkins", Person.Sex.MALE, new DateTime(1922, 05, 27));

    foreach(Person p in pc){
      Console.WriteLine(p.FullName);
    }

    Console.WriteLine("--------- Test the Insert() method -------------------------------");

    pc.Insert(0, p5);

    foreach(Person p in pc){
      Console.WriteLine(p.FullName);
    }

    Console.WriteLine("--------- Test the OnValidate() method -------------------------------");

    try {
      pc.Add(null);
    }catch(ArgumentException){
      Console.WriteLine("Exception thrown attempting to add null value!");
    }
  } // end Main()
} // end class definition
```


Referring to example 18.6 — I first create an instance of PersonCollection followed by several Person objects. I then test the Add() and Remove() methods followed by a test of the read/write capability of the indexer. I then test the Insert() and the OnValidate() methods. You can follow the execution of the code by reading the console output shown in figure 18-2.

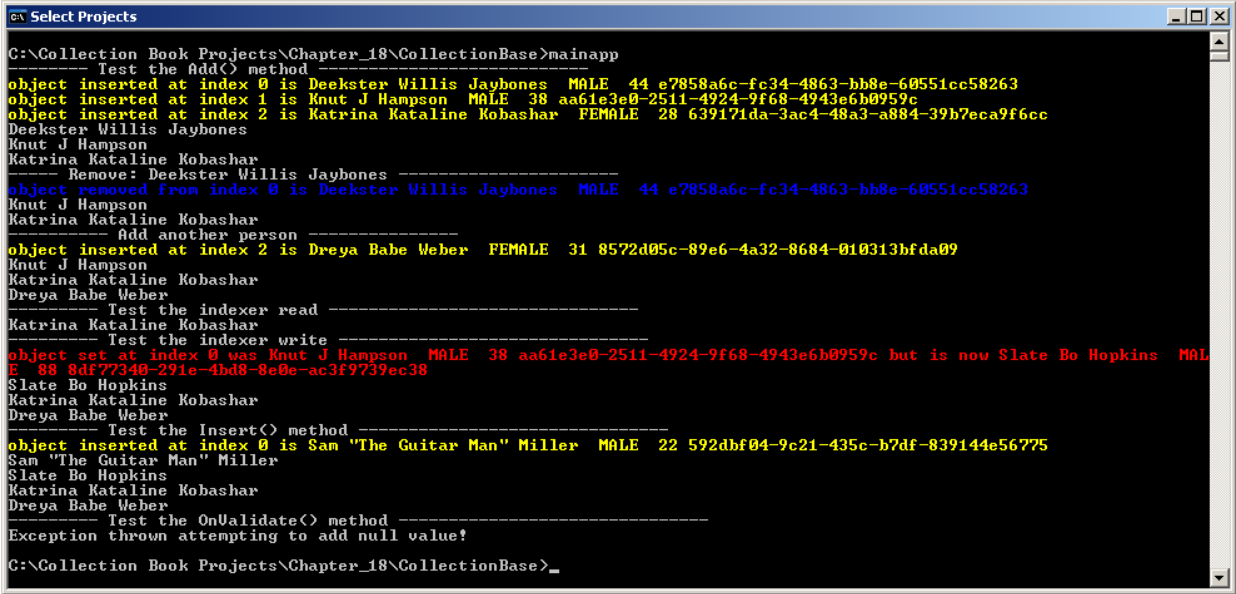


Figure 18-2: Results of Running Example 18.6

Quick Review

You can create a custom collection by extending an existing collection and providing your own implementations of the required class members. The need to create custom, non-generic, strongly-typed collection classes has been rendered an obsolete practice with the introduction of generics. However, not all systems running the .NET framework can update to the latest framework release, so you may encounter legacy code running on production systems that utilize custom, non-generic collections.

Creating A Custom Collection From Scratch

Creating a custom collection from scratch requires a bit of planning. If you want your collection to behave as a collection should behave, you must implement the appropriate interfaces including IEnumerable and IEnumerable<T>. If you need to provide an alternative ordering via an iterator you'll need to implement what are called *named iterators*. You may also need to provide a *custom serialization* process.

In this section I'm going to polish the RedBlackTree class originally introduced in chapter 11. As it stands now the RedBlackTree class is pretty neat and seems to work fine, but I'd like it to behave more like a real collection. By that I mean I want it to implement one of the collection interfaces provided by the .NET collections framework.

First, let's take a look again at the RedBlackTree class and its supporting, custom-developed KeyValuePair and Node classes. The KeyValuePair class is listed in example 18.7.

18.7 KeyValuePair.cs

```
using System;

public class KeyValuePair<TKey, TValue> : IComparable<KeyValuePair<TKey, TValue>> where TKey :
                                                                                   IComparable<TKey> {

  private TKey _key;
  private TValue _value;

  public KeyValuePair(TKey key, TValue value) {
```



```
      _key = key;
      _value = value;
    }

    public KeyValuePair() { }

    public TKey Key {
      get {  return _key; }
      set {  _key = value; }
    }

    public TValue Value {
      get {  return _value; }
      set {  _value = value; }
    }

    public int CompareTo(KeyValuePair<TKey, TValue> other) {
      return this._key.CompareTo(other.Key);
    }

    public override string ToString() {
      return _key.ToString() + " " + _value.ToString();
    }
  } // End KeyValuePair class definition
```

Referring to example 18.7 — the KeyValuePair, as its name implies, represents a key/value pair of objects. The *value* represents the object being inserted into the collection, and the *key* represents the object used to order the value upon insertion. The key object must implement the IComparable<T> interface.

Example 18.8 gives the code for the Node class.

18.8 Node.cs

```
using System;

public class Node<TKey, TValue> where TKey : IComparable<TKey> {

  public KeyValuePair<TKey, TValue> Payload;
  public Node<TKey, TValue> Parent;
  public Node<TKey, TValue> Left;
  public Node<TKey, TValue> Right;

  private bool _color;
  private const bool RED = true;
  private const bool BLACK = false;


  public Node(KeyValuePair<TKey, TValue> payload) {
    Payload = payload;
    _color = RED;
  }

  public bool IsRed {
    get {  return _color; }
  }

  public bool IsBlack {
    get {  return !IsRed; }
  }

  public void MakeRed() {
    _color = RED;

  }

  public void MakeBlack() {
    _color = BLACK;

  }

  public string Color {
    get {  return (_color == RED) ? "RED" : "BLACK"; }
    set {
      switch (value) {
        case "RED": _color = true;
          break;
        case "BLACK": _color = false;
          break;
      }
    }
  }
} // end Node class definition
```


Referring to example 18.8 — the Node class represents a node in the RedBlackTree collection. A node is a data structure that contains a reference to a KeyValuePair payload, a parent node, left node, and a right node. A node can assume the color RED or BLACK.

Example 18.9 lists the code for the RedBlackTree class.

18.9 RedBlackTree.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public class RedBlackTree<TKey, TValue> : IEnumerable where TKey : IComparable<TKey> {


  #region Constants
  private const int EQUALS = 0;
  private const int LESSTHAN = -1;
  private const int GREATERTHAN = 1;
  #endregion

  #region Fields
  private Node<TKey, TValue> _root;
  private int _count = 0;
  private int _left_rotates = 0;
  private int _right_rotates = 0;
  private TKey _first_inserted_key;
  private bool _debug = true;
  #endregion


  #region Constructors
  public RedBlackTree() : this(true) { }

  public RedBlackTree(bool debug) {
    _debug = debug;
  }
  #endregion


  #region Properties
  public KeyValuePair<TKey, TValue> Root {
    get { return _root.Payload; }
  }

  public int Count {
    get { return _count; }
  }
  #endregion


  #region Methods


  /****************************************************************
   *   Insert Method
   * ************************************************************/
  public void Insert(TKey key, TValue value) {
    if ((key == null) || (value == null)) {
      throw new ArgumentException("Invalid Key and/or Value arguments!");
    }
    if (_root == null) {
      _root = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

      _count++;
      if (_debug) {
        Console.WriteLine("Inserted root node with values:" + _root.Payload.ToString());
      }
      _root.MakeBlack();
      _first_inserted_key = _root.Payload.Key;
      return;
    } else {
      Node<TKey, TValue> new_node = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

      bool inserted = false;
      int comparison_result = 0;
      Node<TKey, TValue> node = _root;
      while (!inserted) {
        comparison_result = new_node.Payload.Key.CompareTo(node.Payload.Key);
        switch (comparison_result) {
```



```
        case EQUALS: inserted = true; // ignore duplicate key values
          break;
        case LESSTHAN: if (node.Left == null) {
            node.Left = new_node;
            new_node.Parent = node;
            inserted = true;
            _count++;
            if (_debug) {
              Console.WriteLine("Inserted left: {0}", new_node.Payload.Key);
            }
            RBInsertFixUp(new_node);

          } else {
            node = node.Left;
          }
          break;
        case GREATERTHAN: if (node.Right == null) {
            node.Right = new_node;
            new_node.Parent = node;
            inserted = true;
            _count++;
            if (_debug) {
              Console.WriteLine("Inserted right: {0}", new_node.Payload.Key);
            }
            RBInsertFixUp(new_node);
          } else {
            node = node.Right;
          }
          break;
      }
    }
  }
} // end Insert() method




/**************************************************************
 *  RBInsertFixUp Method
 * ***********************************************************/
private void RBInsertFixUp(Node<TKey, TValue> node) {
  while ((node.Parent != null) && (node.Parent.IsRed)) {
    Node<TKey, TValue> y = null;
    if ((node.Parent.Parent != null) && (node.Parent == node.Parent.Parent.Left)) {
                                             // Parent is a left child
      y = node.Parent.Parent.Right;

      if ((y != null) && (y.IsRed)) {   //case 1
        node.Parent.MakeBlack();        //case 1
        y.MakeBlack();                  //case 1
        node.Parent.Parent.MakeRed();   //case 1
        node = node.Parent.Parent;

        if (node.IsRed) {
          continue;
        }

      } else if (node == node.Parent.Right) {  //case 2
        node = node.Parent;                    //case 2
        RotateLeft(node);                      //case 2
      }

      /**************/
      if ((node.Parent != null)) {           //case 3
        node.Parent.MakeBlack();             //case 3
        if (node.Parent.Parent != null) {    //case 3
          node.Parent.Parent.MakeRed();      //case 3
          RotateRight(node.Parent.Parent);   //case 3
        }
      }
      /*******************/

    } else {                               //Parent is a right child

      if (node.Parent.Parent != null) {
        y = node.Parent.Parent.Left;
      }

      if ((y != null) && (y.IsRed)) {       //case 1
        node.Parent.MakeBlack();            //case 1
        y.MakeBlack();                      //case 1
```



```
        node.Parent.Parent.MakeRed();       //case 1
        node = node.Parent.Parent;

        if (node.IsRed) {
          continue;
        }

      } else if (node == node.Parent.Left) {  //case 2
        node = node.Parent;                   //case 2
        RotateRight(node);                    //case 2
      }

      /*******************/
      if ((node.Parent != null)) {            //case 3
        node.Parent.MakeBlack();              //case 3
        if (node.Parent.Parent != null) {     //case 3
          node.Parent.Parent.MakeRed();       //case 3
          RotateLeft(node.Parent.Parent);     //case 3
        }
      }
      /********************/

    } // end if

    _root.MakeBlack();

  } // end while
} // end RBInsertFixUp() method




/**************************************************************
 * RotateLeft Method
 * ***********************************************************/
private void RotateLeft(Node<TKey, TValue> x) {
  if (x.Right != null) {
    if (_debug) {
      Console.WriteLine("*******************************************************");
      Console.WriteLine("Left Rotate tree  with node x = {0}", x.Payload.Key);
      Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                       (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
      Console.WriteLine("*******************************************************");
    }

    Node<TKey, TValue> y = x.Right;
    if (y != null) {
      x.Right = y.Left;
      if (y.Left != null) {
        y.Left.Parent = x;
      }
      y.Parent = x.Parent;
      if (x.Parent == null) {
        _root = y;
      } else if (x == x.Parent.Left) {
        x.Parent.Left = y;
      } else {
        x.Parent.Right = y;
      }

      y.Left = x;
      x.Parent = y;
    }

    _left_rotates++;
    if (_debug) {
      Console.WriteLine("*******************************************************");
      Console.WriteLine("After left rotate node x = {0}", x.Payload.Key);
      Console.WriteLine("Node x parent = {0}",
                       (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
      Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                       (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
      Console.WriteLine("Left Rotates: {0}", _left_rotates);
      Console.WriteLine("*******************************************************");
    }
  }
} // end RotateLeft() method
```



```
/**************************************************************
 *  RotateRight Method
 * ************************************************************/
private void RotateRight(Node<TKey, TValue> x) {
  if (x.Left != null) {
    if (_debug) {
      Console.WriteLine("*****************************************************");
      Console.WriteLine("Right Rotate tree  with node x = {0}", x.Payload.Key);
      Console.WriteLine("Node color: {0}  Node's parent color: {1}", x.Color,
                        (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
      Console.WriteLine("*****************************************************");
    }

    Node<TKey, TValue> y = x.Left;
    if (y == null) {
      x.Left = null;
    } else {
      x.Left = y.Right;
      if (y.Right != null) {
        y.Right.Parent = x;
      }
      y.Parent = x.Parent;
      if (y.Parent == null) {
        _root = y;
      } else if (x == y.Parent.Left) {
        y.Parent.Left = y;
      } else {
        y.Parent.Right = y;
      }
      y.Right = x;
      x.Parent = y;
    }

    _right_rotates++;
    if (_debug) {
      Console.WriteLine("*****************************************************");
      Console.WriteLine("After right rotate node x = {0}", x.Payload.Key);
      Console.WriteLine("Node x parent = {0}",
                        (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
      Console.WriteLine("Node color: {0}  Node's parent color: {1}", x.Color,
                        (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
      Console.WriteLine("Root = {0}", _root.Payload.Key);
      Console.WriteLine("Right Rotates: {0}", _right_rotates);
      Console.WriteLine("*****************************************************");
    }
  }
} // end RotateRight() method


/**************************************************************
 *  Search Method
 * ************************************************************/
public Node<TKey, TValue> Search(TKey key) {
  int compare_result = 0;
  bool key_found = false;
  Node<TKey, TValue> node = _root;

  while (!key_found) {
    compare_result = key.CompareTo(node.Payload.Key);
    switch (compare_result) {
      case EQUALS: key_found = true;
        break;
      case LESSTHAN: if (node.Left == null) {
          return null;
        }
        node = node.Left;
        break;
      case GREATERTHAN: if (node.Right == null) {
          return null;
        }
        node = node.Right;
        break;

    }
  }
  return node;
}
```



```
/************************************************************
 *  Delete Method
 * **********************************************************/
public void Delete(Node<TKey, TValue> z) {
  if (z == null) return;
  Node<TKey, TValue> y = null;
  if ((z.Left == null) || (z.Right == null)) {
    y = z;
  } else {
    y = TreeSuccessor(z);
  }

  Node<TKey, TValue> x = null;

  if (y.Left != null) {
    x = y.Left;
  } else {
    x = y.Right;
  }
  if (x != null) {
    x.Parent = y.Parent;
  }

  if (y.Parent == null) {
    _root = x;
  } else if (y == y.Parent.Left) {
    y.Parent.Left = x;
  } else {
    y.Parent.Right = x;
  }

  if (y != z) {
    z.Payload = y.Payload;
  }

  if (y.IsBlack) {
    RBDeleteFixUp(x);
  }
  _count--;
}



/************************************************************
 *  RBDeleteFixup
 * **********************************************************/
private void RBDeleteFixUp(Node<TKey, TValue> x) {
  while ((x != null) && (x != _root) && (x.IsBlack)) {
    if (x == x.Parent.Left) {
      Node<TKey, TValue> w = x.Parent.Right;
      if ((w != null) && w.IsRed) {
        w.MakeBlack();
        x.Parent.MakeRed();
        RotateLeft(x.Parent);
        w = x.Parent.Right;
      }

      if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                      && ((w.Right == null) || w.Right.IsBlack)) {
        w.MakeRed();
        x = x.Parent;
        continue;

      } else if ((w != null) && w.Right.IsBlack) {
        w.Left.MakeBlack();
        w.MakeRed();
        RotateRight(w);
        w = x.Parent.Right;
      }

      /**********************/
      if (w != null) {
        w.Color = x.Parent.Color;
        x.Parent.MakeBlack();
        w.Right.MakeBlack();

      }
      RotateLeft(x.Parent);
      x = _root;
      /**********************/

```

```
      } else {

        Node<TKey, TValue> w = x.Parent.Left;
        if ((w != null) && w.IsRed) {
          w.MakeBlack();
          x.Parent.MakeRed();
          RotateRight(x.Parent);
          w = x.Parent.Left;
        }

        if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                        && ((w.Right == null) || w.Right.IsBlack)) {
          w.MakeRed();
          x = x.Parent;
          continue;

        } else if ((w != null) && w.Left.IsBlack) {
          w.Right.MakeBlack();
          w.MakeRed();
          RotateLeft(w);
          w = x.Parent.Left;
        }

        /***********************/
        if (w != null) {
          w.Color = x.Parent.Color;
          x.Parent.MakeBlack();
          w.Right.MakeBlack();

        }
        RotateRight(x.Parent);
        x = _root;
        /**********************/
      }

      x.MakeBlack();
      _root.MakeBlack();
    } // end while

  } // end RBDeleteFixUp



  /***************************************************************
   * TreeSuccessor Method
   * *************************************************************/
  private Node<TKey, TValue> TreeSuccessor(Node<TKey, TValue> node) {
    if (node.Right != null) {
      return TreeMinimum(node.Right);
    }
    Node<TKey, TValue> y = node.Parent;
    while ((y != null) && (node == y.Right)) {
      node = y;
      y = y.Parent;
    }
    return y;
  }


  /***************************************************************
   * TreeMinimum Method
   * *************************************************************/
  private Node<TKey, TValue> TreeMinimum(Node<TKey, TValue> node) {
    while (node.Left != null) {
      node = node.Left;
    }
    return node;
  }


  /***************************************************************
   *  TreeMaximum Method -- Not used in this program
   * *************************************************************/
  private Node<TKey, TValue> TreeMaximum(Node<TKey, TValue> node) {
    while (node.Right != null) {
      node = node.Right;
    }
    return node;
  }


```



```
479   /*************************************************************
480    *  GetEnumerator Method
481    * ***********************************************************/
482   public IEnumerator GetEnumerator() {
483     return this.ToArray().GetEnumerator();
484   }
485
486
487   /*************************************************************
488    *  ToArray Method
489    * ***********************************************************/
490   public KeyValuePair<TKey, TValue>[] ToArray() {
491     KeyValuePair<TKey, TValue>[] _items = new KeyValuePair<TKey, TValue>[_count];
492     int index = 0;
493     this.WalkTree(_root, _items, ref index);
494     return _items;
495   }
496
497
498   /*************************************************************
499    *  WalkTree Method
500    * ***********************************************************/
501   private void WalkTree(Node<TKey, TValue> node, KeyValuePair<TKey, TValue>[] items, ref int index) {
502     if (node != null) {
503       WalkTree(node.Left, items, ref index);
504       items[index++] = node.Payload;
505       if (_debug) {
506         if (node == _root) {
507           Console.WriteLine("**********ROOT NODE: {0}:{1}**********",
508                             node.Payload.Value, node.Color);
509         } else {
510           Console.WriteLine("Walking Tree - Node visited: {0} Color: {1}",
511                             node.Payload.Value, node.Color);
512         }
513       }
514       WalkTree(node.Right, items, ref index);
515     }
516   }
517
518
519   /*************************************************************
520    * PrintTreeToConsole Method
521    * ***********************************************************/
522   public void PrintTreeToConsole() {
523     foreach (KeyValuePair<TKey, TValue> item in this) {
524       if (item.Key.CompareTo(_root.Payload.Key) == 0) {
525         Console.ForegroundColor = ConsoleColor.Yellow;
526         Console.Write(item.Key + " ");
527         Console.ForegroundColor = ConsoleColor.White;
528       } else {
529         Console.Write(item.Key + " ");
530       }
531     }
532     Console.WriteLine();
533   }
534
535
536   /*************************************************************
537    *  PrintTreeStats Method
538    * ***********************************************************/
539   public void PrintTreeStats() {
540     Console.WriteLine("------------ Tree Stats --------------------");
541     Console.WriteLine("First inserted key: {0}", _first_inserted_key);
542     Console.WriteLine("Count: {0}", _count);
543     Console.WriteLine("Left Rotates: {0}", _left_rotates);
544     Console.WriteLine("Right Rotates: {0}", _right_rotates);
545     Console.WriteLine("--------------------------------------------");
546   }
547
548   #endregion
549
550 } // end RedBlackTree class
```

Referring to example 18.9 — the RedBlackTree class implements the IEnumerable interface. The GetEnumerator() method on line 482 simply converts the RedBlackTree into an array with a call to its ToArray() method and in turn calls the GetEnumerator() method on the array. While it works, it's a bit of a hack.

Evaluating RedBlackTree

Although RedBlackTree implements IEnumerable, it is not formally a collection class because it doesn't implement an interface that tags it as being a collection. And while it implements the IEnumerable interface, it doesn't implement IEnumerable<T>, so the job is only half done.

At this point I have not tried to persist or serialize a populated instance of RedBlackTree, but the first step in doing so would be to apply the [Serializable] attribute to it and its supporting classes and give it a try.

Also, the custom KeyValuePair class is unnecessary and redundant. The .NET framework already provides a generic KeyValuePair class. I can exchange my version of the KeyValuePair class for the .NET framework version and feel confident I'm getting a better version of that piece of code.

Selecting The Appropriate Interface

In its current state, the RedBlackTree class has the inheritance hierarchy shown in figure 18-3.

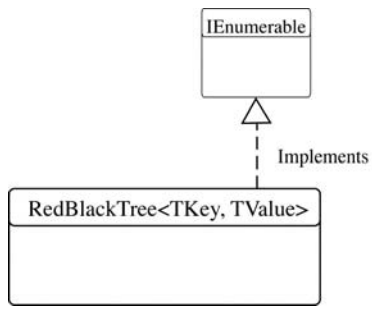


Figure 18-3: RedBlackTree Inheritance Diagram — Current State of Affairs

Referring to figure 18-3 — although the RedBlackTree class implements the IEnumerable interface and can be enumerated with the `foreach` statement, that's not enough to make it a collection.

The closest and most appropriate interface that represents the operations and functionality available in a RedBlackTree is the IDictionary<TKey, TValue> interface. Implementing the IDictionary<TKey, TValue> interface will radically change the *public behavior* of the RedBlackTree, making it more a collection and less a one-off data structure. The term *public behavior* refers to the public members exposed by the class.

Figure 18-4 offers a revised UML diagram for the planned modification. Referring to figure 18-4 — implementing the IDictionary<TKey, TValue> interface will result in a more robust class and force the consideration of its behavior as a collection as opposed to a one-off custom data structure. Remember when studying figure 18-4 that an interface can extend another interface. The behavior declared in the IDictionary<KeyValuePair<TKey, TValue>> interface is the sum of the functionality declared in the interfaces of its inheritance hierarchy.

Implementing IEnumerable and IEnumerable<T>

Note again by referring to figure 18-4 that implementing the IDictionary<TKey, TValue> interface will also force the implementation of both the IEnumerable and IEnumerable<T> interfaces. The implementation of these interfaces will provide a default enumeration for the RedBlackTree. Recall that the purpose of enumerators and the `foreach` statement is to allow users to iterate over the elements of a collection in a uniform way regardless of the structure of the collection.

Named Iterators

If you need to provide an alternative way to iterate over your collection with the `foreach` statement you'll need to implement one or more *named iterators*. The default ordering provided by the straight forward implementation of the IEnumerable and IEnumerable<T> interface for the RedBlackTree collection might offer up its elements in ascending order, but what if you wanted to iterate over the collection in descending order? A named iterator will allow you to do this.

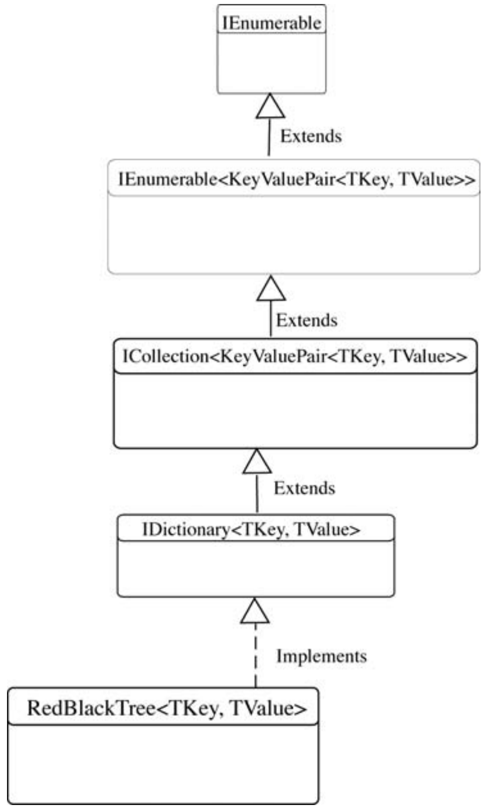


Figure 18-4: RedBlackTree Inheritance Diagram after Implementing IDictionary<TKey, TValue> Interface

Serialization Considerations

If you want to persist a custom collection using binary serialization, everything should work fine as long as the collection and all the objects stored in the collection are tagged with the [Serializable] attribute. The only sure way to know is to give it a try. The RedBlackTree collection works flawlessly with binary serialization.

The Need for Custom Serialization

If you need to persist a custom collection in a format other than a binary representation, you'll need to implement a custom serialization solution. For example, to serialize a RedBlackTree in XML format you will need to implement the IXmlSerializable interface and implement the WriteXml(), ReadXml(), and GetSchema() methods. Tag each property you wish to persist in your user-defined data types with the [XmlElement] attribute.

Responding to Collection Changing Events

To respond to collection changing events like adding or removing elements you'll need to implement the INotifyCollectionChanged interface.

Extended Example: The RedBlackTree Collection

The following examples provide a complete implementation of the modified RedBlackTree class. In this example I'm using a RedBlackTree to store Person objects inserted using a PersonKey. I then serialize the collection using an XmlSerializer. Let's start with the Person class.

18.10 Person.cs

```
using System;
using System.ComponentModel;
using System.Xml.Serialization;
using System.Xml.Schema;
using System.Xml;

[Serializable]

public class Person : IComparable, IComparable<Person>, INotifyPropertyChanged {


  //enumeration
  public enum Sex {MALE, FEMALE};


  //event
  public event PropertyChangedEventHandler PropertyChanged;


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){
    _firstName = string.Empty;
    _middleName = string.Empty;
    _lastName = string.Empty;
    _gender = Person.Sex.MALE;
    _birthday = DateTime.Now;
    _dna = Guid.NewGuid();

  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  [XmlElement]
  public String FirstName {
    get { return _firstName; }
    set { _firstName = value;
          NotifyPropertyChanged("FirstName");
        }
  }

  [XmlElement]
  public String MiddleName {
```



```
      get {  return _middleName; }
      set {  _middleName = value;
            NotifyPropertyChanged("MiddleName");
         }
   }

   [ XmlElement]
   public String LastName {
      get {  return _lastName; }
      set {  _lastName = value;
            NotifyPropertyChanged("LastName");
         }
   }

   [ XmlElement]
   public Sex Gender {
      get {  return _gender; }
      set {  _gender = value;
            NotifyPropertyChanged("Gender");
         }
   }

   [ XmlElement]
   public DateTime Birthday {
      get {  return _birthday; }
      set {  _birthday = value;
            NotifyPropertyChanged("Birthday");
         }
   }

   [ XmlElement]
   public Guid DNA {
      get {  return _dna; }
      set {  _dna = value;
            NotifyPropertyChanged("DNA");
      }
   }

   public int Age {
       get {
        int years = DateTime.Now.Year - _birthday.Year;
         int adjustment = 0;
        if(DateTime.Now.Month < _birthday.Month){
           adjustment = 1;
         }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
             adjustment = 1;
         }
      return years - adjustment;
      }
    }

   public String FullName {
      get {  return FirstName + " " + MiddleName + " " + LastName; }
   }

   public String FullNameAndAge {
      get {  return FullName + " " + Age; }
   }

   protected String SortableName {
      get {  return LastName + FirstName + MiddleName; }
   }

   public PersonKey Key {
      get {  return new PersonKey(this.ToString()); }
   }

   public override String ToString(){
      return (FullName + "  " + Gender + "  " + Age + " " + DNA);
   }

   public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(Person) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
   }

   public override int GetHashCode(){
      return this.ToString().GetHashCode();
   }

```



```
  public static bool operator ==(Person lhs, Person rhs){
    return lhs.Equals(rhs);
  }

  public static bool operator !=(Person lhs, Person rhs){
    return !(lhs.Equals(rhs));
  }

  public int CompareTo(object obj){
    if((obj == null) || (typeof(Person) != obj.GetType()))  {
      throw new ArgumentException("Object is not a Person!");
    }
    return this.SortableName.CompareTo(((Person)obj).SortableName);
  }

  public int CompareTo(Person p){
    if(p == null){
      throw new ArgumentException("Cannot compare null objects!");
    }
    return this.SortableName.CompareTo(p.SortableName);
  }


  private void NotifyPropertyChanged(string propertyName){
    if(PropertyChanged != null){
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
  }

} // end Person class
```

Referring to example 18.10 — I've modified the Person class by adding the [XmlElement] attribute above each property I want to persist using XmlSerialization. Note that the use of the [XmlElement] in this fashion is not strictly necessary since I'm interested only in persisting the public read/write properties of a Person object in a straight forward fashion. For example, if I didn't apply the [XmlAttribute] to the FirstName property it would, by default, be saved as an XML element with the tags `<FirstName> </FirstName>`.

18.11 PersonKey.cs

```
using System;

[Serializable]
public class PersonKey : IEquatable<String>, IComparable, IComparable<PersonKey> {

    private string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public PersonKey(){}

    public String KeyString {
      get {  return _keyString; }
      set {  _keyString = value; }
    }

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

    public int CompareTo(object obj){
     return _keyString.CompareTo(obj);
    }


    public int CompareTo(PersonKey pk){
```



```
43        return _keyString.CompareTo(pk._keyString);
44      }
45  }
```

Referring to example 18.11 — I've modified the PersonKey class by adding a read/write property named KeyString which begins on line 14. I've also removed the `readonly` declaration from the _keyString field. I did this so I could recreate the PersonKey when reading it from an XML file.

18.12 Node.cs

```
using System;
using System.Collections.Generic;

[Serializable]
public class Node<TKey, TValue> where TKey : IComparable<TKey> {

  public KeyValuePair<TKey, TValue> Payload;
  public Node<TKey, TValue> Parent;
  public Node<TKey, TValue> Left;
  public Node<TKey, TValue> Right;

  private bool _color;
  private const bool RED = true;
  private const bool BLACK = false;


  public Node(KeyValuePair<TKey, TValue> payload) {
    Payload = payload;
    _color = RED;
  }

  public bool IsRed {
    get {  return _color; }
  }

  public bool IsBlack {
    get {  return !IsRed; }
  }

  public void MakeRed() {
    _color = RED;

  }

  public void MakeBlack() {
    _color = BLACK;

  }

  public string Color {
    get {  return (_color == RED) ? "RED" : "BLACK"; }
    set {
      switch (value) {
        case "RED": _color = true;
          break;
        case "BLACK": _color = false;
          break;
      }
    }
  }
}
```

Referring to example 18.12 — the Node class remains unchanged. It represents a node in a RedBlackTree. It has public left, right, and parent fields and can assume the color RED or BLACK.

18.13 RedBlackTree.cs

```
using System;
using System.Collections;
using System.Collections.ObjectModel;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Linq;
using System.Xml.Serialization;
using System.Xml.Schema;
using System.Xml;


[Serializable]
public class RedBlackTree<TKey, TValue> : IDictionary<TKey, TValue>, INotifyCollectionChanged,
                                          IXmlSerializable where TKey : IComparable<TKey> {

  #region Constants
  private const int EQUALS = 0;
```



```
   private const int LESSTHAN = -1;
   private const int GREATERTHAN = 1;
   private const string DEBUG = "Debug";
   private const string KEY_VALUE_PAIR = "KeyValuePair";
   private const string KEY = "Key";
   private const string VALUE = "Value";
   #endregion

   #region Fields
   private Node<TKey, TValue> _root;
   private int _count = 0;
   private int _left_rotates = 0;
   private int _right_rotates = 0;
   private TKey _first_inserted_key;
   private bool _debug = false;
   #endregion

   #region Events
     public event NotifyCollectionChangedEventHandler CollectionChanged;
   #endregion


   #region Constructors
  /*****************************************************
     *  Default Constructor - Sets debug to true by default
     ****************************************************/
   public RedBlackTree() : this(false) { }


   /*************************************************
     *  Single argument constructor
     ****************************************************/
   public RedBlackTree(bool debug) {
     _debug = debug;
   }
   #endregion


   #region Properties
   public Node<TKey, TValue> Root {
     get {  return _root; }
   }

   public int Count {
     get {  return _count; }
   }

   public bool IsReadOnly {
     get {  return false; }
   }

   public bool Debug {
     get {  return _debug; }
     set {  _debug = value; }
   }

   /***********************************************************
     *  Indexer - Raises NotifiyCollectionChangedEvent when setter is accessed.
     ***********************************************************/
   public TValue this[ TKey key]{

     get {
       if(key == null){
         throw new ArgumentNullException("Key is null!");
       }
       Node<TKey, TValue> temp = this.Search(key);
       if(temp == null){
         throw new KeyNotFoundException("Key not found!");
       }
       return temp.Payload.Value;
     }

     set {
       if(key == null){
         throw new ArgumentNullException("Key is null!");
       }
       if(value == null){
          throw new ArgumentNullException("Value is null!");
       }
       Node<TKey, TValue> temp = this.Search(key);
       if(temp == null){
```



```
      this.Add(key, value);
      //Raise the collection changed event...
      // First, create the NotifyCollectionChangedEventArgs object
      NotifyCollectionChangedEventArgs args =
        new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add,
                                                     (this.Search(key)).Payload);
      //Then pass it along to the NotifyCollectionChanged() method...
      this.NotifyCollectionChanged(args);
    }else{

       this.Remove(key);
       this.Add(key, value);
       //Raise the collection changed event...
       // First, create the NotifyCollectionChangedEventArgs object
       NotifyCollectionChangedEventArgs args =
         new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Replace,
                                                  (this.Search(key)).Payload, temp);
       //Then pass it along to the NotifyCollectionChanged() method...
       this.NotifyCollectionChanged(args);
    }

  }
}

public ICollection<TKey> Keys {
  get { return this.GetKeyCollection(); }
}

public ICollection<TValue> Values {
  get { return this.GetValueCollection(); }
}

#endregion


#region IDictionary<TKey, TValue> Interface Methods

/**********************************************************
  * Add() - single argument method. Passes call to this.Insert().
  **********************************************************/
public void Add(KeyValuePair<TKey, TValue> item){
  this.Insert(item.Key, item.Value);
}

/**********************************************************
  * Add() - double argument method. Passes call to this.Insert().
  **********************************************************/
public void Add(TKey key, TValue value){
  if((key == null) || (value == null)){
    throw new ArgumentNullException("Tried to add null item to collection...");
  }
  this.Insert(key, value);
}


/**********************************************************
  * Clear() - Passes call to this.ClearTree() method.
  **********************************************************/
public void Clear(){
  this.ClearTree(_root);
  //Raise the collection changed event...
  // First, create the NotifyCollectionChangedEventArgs object
  NotifyCollectionChangedEventArgs args =
  new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Reset);
  //Then pass it along to the NotifyCollectionChanged() method...
  this.NotifyCollectionChanged(args);
}


/**********************************************************
  * Contains() - finds item via this.Search() method
  **********************************************************/
public bool Contains(KeyValuePair<TKey, TValue> item){
  Node<TKey, TValue> temp = this.Search(item.Key);
  bool return_value = false;
  if(temp == null){
    return_value = false;
  }else{
    return_value = true;
  }
  return return_value;
```



```
}

/*******************************************************
  * ContainsKey() - finds item via this.Search() method.
  *******************************************************/
public bool ContainsKey(TKey key){
   if(key == null){
     throw new ArgumentNullException("Cannot search for a null key!");
   }
   Node<TKey, TValue> result = this.Search(key);
   if(result == null){
     return false;
   }
   return result.Payload.Key.Equals(key);
}


/*******************************************************
  * CopyTo() - Copies tree elements to an array.
  *******************************************************/
public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex){
   if(array == null){
     throw new ArgumentNullException("Array argument is null!");
   }
   if(array.Rank > 1){
     throw new ArgumentException("Array must be single dimensional!");
   }
   if(arrayIndex < 0){
     throw new ArgumentOutOfRangeException("Index argument is less than zero!");
   }
   if(((array.Length - arrayIndex)-1) < this.Count){
     throw new ArgumentException("Not enought space in array to contain collection items!");
   }

   foreach(KeyValuePair<TKey, TValue> kvp in this.ToArray()){
      array[ arrayIndex++] = kvp;
   }
}

/*************************************************************
  * Remove() - Removes an element from the tree. Passes call to this.Remove()
  *************************************************************/
public bool Remove(KeyValuePair<TKey, TValue> item){
   if(item.Key == null){
    throw new ArgumentNullException("Item.Key is null!");
  }
  return this.Remove(item.Key);

}

/**************************************************************
  * Remove() - Removes an item with specified key. Passes call to this.Delete()
  **************************************************************/
public bool Remove(TKey key){
  if(key == null){
    throw new ArgumentNullException("Key argument is null!");
  }
  bool result = false;
  Node<TKey, TValue> temp = this.Search(key);
  if(temp != null){
    this.Delete(temp);
    result = true;
  }
  return result;
}


/*****************************************************************
  * TryGetValue() - Searches for value and returns true if in tree, false otherwise.
  *****************************************************************/
public bool TryGetValue(TKey key, out TValue value){
  if(key == null){
    throw new ArgumentNullException("Key is null!");
  }
  bool result = false;
  Node<TKey, TValue> temp = this.Search(key);
  TValue ret_value = default(TValue);
    if(temp == null){
      result = false;
    } else{
      ret_value = temp.Payload.Value;
```



```
      result = true;
     }
  value = ret_value;
  return result;
}

#endregion



#region Specialized Methods


/*****************************************************************
 *   Insert()  method
 * ************************************************************/
public void Insert(TKey key, TValue value) {
  if ((key == null) || (value == null)) {
    throw new ArgumentException("Invalid Key and/or Value arguments!");
  }

  NotifyCollectionChangedEventArgs args = null;

  if (_root == null) {
    _root = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

    _count++;
    if (_debug) {
      Console.WriteLine("Inserted root node with values:" + _root.Payload.ToString());
    }
    _root.MakeBlack();
    _first_inserted_key = _root.Payload.Key;
    //Raise the collection changed event...
    // First, create the NotifyCollectionChangedEventArgs object
    args = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add, _root.Payload);
    //Then pass it along to the NotifyCollectionChanged() method...
    this.NotifyCollectionChanged(args);
    return;
  } else {
    Node<TKey, TValue> new_node = new Node<TKey, TValue>(new KeyValuePair<TKey, TValue>(key, value));

    bool inserted = false;
    int comparison_result = 0;
    Node<TKey, TValue> node = _root;
    while (!inserted) {
      comparison_result = new_node.Payload.Key.CompareTo(node.Payload.Key);
      switch (comparison_result) {
        case EQUALS: inserted = true; // ignore duplicate key values
          break;
        case LESSTHAN: if (node.Left == null) {
            node.Left = new_node;
            new_node.Parent = node;
            inserted = true;
            _count++;
            if (_debug) {
              Console.WriteLine("Inserted left: {0}", new_node.Payload.Key);
            }
            RBInsertFixUp(new_node);

          } else {
            node = node.Left;
          }
          break;
        case GREATERTHAN: if (node.Right == null) {
            node.Right = new_node;
            new_node.Parent = node;
            inserted = true;
            _count++;
            if (_debug) {
              Console.WriteLine("Inserted right: {0}", new_node.Payload.Key);
            }
            RBInsertFixUp(new_node);
          } else {
            node = node.Right;
          }
          break;
      }
    }

    //Raise the collection changed event...
    // First, create the NotifyCollectionChangedEventArgs object
```



```
    args = new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add, new_node.Payload);
    //Then pass it along to the NotifyCollectionChanged() method...
    this.NotifyCollectionChanged(args);
  }
} // end Insert() method




/***************************************************************
 *  RBInsertFixUp()  method
 * ************************************************************/
private void RBInsertFixUp(Node<TKey, TValue> node) {
  while ((node.Parent != null) && (node.Parent.IsRed)) {
    Node<TKey, TValue> y = null;
    if ((node.Parent.Parent != null) && (node.Parent == node.Parent.Parent.Left)) {
       // Parent is a left child
      y = node.Parent.Parent.Right;

      if ((y != null) && (y.IsRed)) {   //case 1
        node.Parent.MakeBlack();        //case 1
        y.MakeBlack();                  //case 1
        node.Parent.Parent.MakeRed();   //case 1
        node = node.Parent.Parent;

        if (node.IsRed) {
          continue;
        }

      } else if (node == node.Parent.Right) {  //case 2
        node = node.Parent;                    //case 2
        RotateLeft(node);                      //case 2
      }

      /**************/
      if ((node.Parent != null)) {            //case 3
        node.Parent.MakeBlack();              //case 3
        if (node.Parent.Parent != null) {     //case 3
          node.Parent.Parent.MakeRed();       //case 3
          RotateRight(node.Parent.Parent);    //case 3
        }
      }
      /********************/

    } else {                                //Parent is a right child

      if (node.Parent.Parent != null) {
        y = node.Parent.Parent.Left;
      }

      if ((y != null) && (y.IsRed)) {       //case 1
        node.Parent.MakeBlack();            //case 1
        y.MakeBlack();                      //case 1
        node.Parent.Parent.MakeRed();       //case 1
        node = node.Parent.Parent;

        if (node.IsRed) {
          continue;
        }

      } else if (node == node.Parent.Left) {   //case 2
        node = node.Parent;                    //case 2
        RotateRight(node);                     //case 2
      }

      /********************/
      if ((node.Parent != null)) {             //case 3
        node.Parent.MakeBlack();               //case 3
        if (node.Parent.Parent != null) {      //case 3
          node.Parent.Parent.MakeRed();        //case 3
          RotateLeft(node.Parent.Parent);      //case 3
        }
      }
      /*********************/

    } // end if

    _root.MakeBlack();

  } // end while
} // end RBInsertFixUp() method
```



```
/**************************************************************
 * RotateLeft()  method
 * ************************************************************/
private void RotateLeft(Node<TKey, TValue> x) {
  if (x.Right != null) {
    if (_debug) {
      Console.WriteLine("******************************************************");
      Console.WriteLine("Left Rotate tree  with node x = {0}", x.Payload.Key);
      Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                        (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
      Console.WriteLine("******************************************************");
    }

    Node<TKey, TValue> y = x.Right;
    if (y != null) {
      x.Right = y.Left;
      if (y.Left != null) {
        y.Left.Parent = x;
      }
      y.Parent = x.Parent;
      if (x.Parent == null) {
        _root = y;
      } else if (x == x.Parent.Left) {
        x.Parent.Left = y;
      } else {
        x.Parent.Right = y;
      }

      y.Left = x;
      x.Parent = y;
    }

    _left_rotates++;
    if (_debug) {
      Console.WriteLine("******************************************************");
      Console.WriteLine("After left rotate node x = {0}", x.Payload.Key);
      Console.WriteLine("Node x parent = {0}",
                        (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
      Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                        (x.Parent != null) ? x.Parent.Color.ToString() : "x.Parent is null");
      Console.WriteLine("Left Rotates: {0}", _left_rotates);
      Console.WriteLine("******************************************************");
    }
  }
} // end RotateLeft() method


/**************************************************************
 *  RotateRight()  method
 * ************************************************************/
private void RotateRight(Node<TKey, TValue> x) {
  if (x.Left != null) {
    if (_debug) {
      Console.WriteLine("******************************************************");
      Console.WriteLine("Right Rotate tree  with node x = {0}", x.Payload.Key);
      Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                        (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
      Console.WriteLine("******************************************************");
    }

    Node<TKey, TValue> y = x.Left;
    if (y == null) {
      x.Left = null;
    } else {
      x.Left = y.Right;
      if (y.Right != null) {
        y.Right.Parent = x;
      }
      y.Parent = x.Parent;
      if (y.Parent == null) {
        _root = y;
      } else if (x == y.Parent.Left) {
        y.Parent.Left = y;
      } else {
        y.Parent.Right = y;
      }
```



```
        y.Right = x;
        x.Parent = y;
      }

      _right_rotates++;
      if (_debug) {
        Console.WriteLine("****************************************************");
        Console.WriteLine("After right rotate node x = {0}", x.Payload.Key);
        Console.WriteLine("Node x parent = {0}",
                         (x.Parent != null) ? x.Parent.Payload.Key.ToString() : "x parent is null");
        Console.WriteLine("Node color: {0} Node's parent color: {1}", x.Color,
                         (x.Parent != null) ? x.Parent.Color.ToString() : "X.Parent == null");
        Console.WriteLine("Root = {0}", _root.Payload.Key);
        Console.WriteLine("Right Rotates: {0}", _right_rotates);
        Console.WriteLine("****************************************************");
      }

    }
  } // end RotateRight() method


  /**************************************************************
   *  Search()  method
   * *************************************************************/
  public Node<TKey, TValue> Search(TKey key) {
    int compare_result = 0;
    bool key_found = false;
    Node<TKey, TValue> node = _root;

    while (!key_found) {
      compare_result = key.CompareTo(node.Payload.Key);
      switch (compare_result) {
        case EQUALS: key_found = true;
          break;
        case LESSTHAN: if (node.Left == null) {
            return null;
          }
          node = node.Left;
          break;
        case GREATERTHAN: if (node.Right == null) {
            return null;
          }
          node = node.Right;
          break;

      }
    }

    return node;
  }


  /**************************************************************
   *  Delete()  method
   * *************************************************************/
  public void Delete(Node<TKey, TValue> z) {
    if (z == null) return;
    //Raise the collection changed event...
    // First, create the NotifyCollectionChangedEventArgs object
    NotifyCollectionChangedEventArgs args =
      new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Remove, z.Payload);
    //Then pass it along to the NotifyCollectionChanged() method...
    this.NotifyCollectionChanged(args);
    Node<TKey, TValue> y = null;
    if ((z.Left == null) || (z.Right == null)) {
      y = z;
    } else {
      y = TreeSuccessor(z);
    }

    Node<TKey, TValue> x = null;

    if (y.Left != null) {
      x = y.Left;
    } else {
      x = y.Right;
    }
    if (x != null) {
      x.Parent = y.Parent;
    }

```



```
    if (y.Parent == null) {
      _root = x;
    } else if (y == y.Parent.Left) {
      y.Parent.Left = x;
    } else {
      y.Parent.Right = x;
    }

    if (y != z) {
      z.Payload = y.Payload;
    }

    if (y.IsBlack) {
      RBDeleteFixUp(x);
    }
    _count--;

  }



  /**************************************************************
   *  RBDeleteFixup() method
   * ************************************************************/
  private void RBDeleteFixUp(Node<TKey, TValue> x) {
    while ((x != null) && (x != _root) && (x.IsBlack)) {
      if (x == x.Parent.Left) {
        Node<TKey, TValue> w = x.Parent.Right;
        if ((w != null) && w.IsRed) {
          w.MakeBlack();
          x.Parent.MakeRed();
          RotateLeft(x.Parent);
          w = x.Parent.Right;
        }

        if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                        && ((w.Right == null) || w.Right.IsBlack)) {
          w.MakeRed();
          x = x.Parent;
          continue;

        } else if ((w != null) && w.Right.IsBlack) {
          w.Left.MakeBlack();
          w.MakeRed();
          RotateRight(w);
          w = x.Parent.Right;
        }

        /***********************/
        if (w != null) {
          w.Color = x.Parent.Color;
          x.Parent.MakeBlack();
          w.Right.MakeBlack();

        }
        RotateLeft(x.Parent);
        x = _root;
        /**********************/

      } else {

        Node<TKey, TValue> w = x.Parent.Left;
        if ((w != null) && w.IsRed) {
          w.MakeBlack();
          x.Parent.MakeRed();
          RotateRight(x.Parent);
          w = x.Parent.Left;
        }

        if ((w != null) && ((w.Left == null) || w.Left.IsBlack)
                        && ((w.Right == null) || w.Right.IsBlack)) {
          w.MakeRed();
          x = x.Parent;
          continue;

        } else if ((w != null) && w.Left.IsBlack) {
          w.Right.MakeBlack();
          w.MakeRed();
          RotateLeft(w);
          w = x.Parent.Left;
        }
```



```

      /***********************/
      if (w != null) {
        w.Color = x.Parent.Color;
        x.Parent.MakeBlack();
        w.Right.MakeBlack();

      }
      RotateRight(x.Parent);
      x = _root;
      /**********************/
    }

    x.MakeBlack();
    _root.MakeBlack();
  } // end while

} // end RBDeleteFixUp



/************************************************************
 * TreeSuccessor()  method
 * *********************************************************/
private Node<TKey, TValue> TreeSuccessor(Node<TKey, TValue> node) {
  if (node.Right != null) {
    return TreeMinimum(node.Right);
  }
  Node<TKey, TValue> y = node.Parent;
  while ((y != null) && (node == y.Right)) {
    node = y;
    y = y.Parent;
  }
  return y;
}


/************************************************************
 * TreeMinimum()  method
 * *********************************************************/
private Node<TKey, TValue> TreeMinimum(Node<TKey, TValue> node) {
  while (node.Left != null) {
    node = node.Left;
  }
  return node;
}


/************************************************************
 *  TreeMaximum()  method -- Not used in this program
 * *********************************************************/
private Node<TKey, TValue> TreeMaximum(Node<TKey, TValue> node) {
  while (node.Right != null) {
    node = node.Right;
  }
  return node;
}



/************************************************************
 *  ToArray()  method
 * *********************************************************/
public KeyValuePair<TKey, TValue>[] ToArray() {
  KeyValuePair<TKey, TValue>[] _items = new KeyValuePair<TKey, TValue>[_count];
  int index = 0;
  this.WalkTree(_root, _items, ref index);
  return _items;

}


/********************************************************
   *  GetKeyCollection() method
   *********************************************************/
private ICollection<TKey> GetKeyCollection(){
  Collection<TKey> keys = new Collection<TKey>();
  foreach(KeyValuePair<TKey, TValue> kvp in this.ToArray()){
    keys.Add(kvp.Key);
  }
  return keys;
```



```
  }


  /**********************************************************
     * GetValueCollection() method
     **********************************************************/
  private ICollection<TValue> GetValueCollection(){
    Collection<TValue> values = new Collection<TValue>();
    foreach(KeyValuePair<TKey, TValue> kvp in this.ToArray()){
      values.Add(kvp.Value);
    }
    return values;
  }


  /************************************************************
     *  WalkTree()  method
     * ***********************************************************/
  private void WalkTree(Node<TKey, TValue> node, KeyValuePair<TKey, TValue>[] items, ref int index) {
    if (node != null) {
      WalkTree(node.Left, items, ref index);
      items[index++] = node.Payload;
      if (_debug) {
        if (node == _root) {
          Console.WriteLine("**********ROOT NODE: {0}:{1}**********",
                            node.Payload.Value, node.Color);
        } else {
          Console.WriteLine("Walking Tree - Node visited: {0} Color: {1}",
                            node.Payload.Value, node.Color);
        }
      }
      WalkTree(node.Right, items, ref index);
    }
  }


  /*********************************************************
    *  Default IEnumerable GetEnumerator() method
    *********************************************************/
  IEnumerator IEnumerable.GetEnumerator(){
      return this.ToArray().GetEnumerator();
  }


  /*********************************************************
      * Default IEnumerable<T> GetEnumerator() method
      *********************************************************/
  public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator(){
    KeyValuePair<TKey, TValue>[] items = new KeyValuePair<TKey, TValue>[_count];
    int index = 0;
    InOrderTraversalIterator(_root, items, ref index);
    for(int i = 0; i < items.Length; i++){
      yield return items[i];
    }
  }


  /*********************************************************
    * HighToLow named iterator property
    *********************************************************/
  public IEnumerable<KeyValuePair<TKey, TValue>> HighToLow {

    get {
      KeyValuePair<TKey, TValue>[] items = new KeyValuePair<TKey, TValue>[_count];
      int index = 0;
      PostOrderTraversal(_root, items, ref index);
      for(int i = 0; i < items.Length; i++){
        yield return items[i];
      }
    }
  }


 /***********************************************************
  *  In-Order Traversal method -- used for the default iteration
  ***********************************************************/
  private void InOrderTraversalIterator(Node<TKey, TValue> node, KeyValuePair<TKey, TValue>[] items,
                                                                                      ref int index){
     if(node != null){
        InOrderTraversalIterator(node.Left, items, ref index);
         items[index++] = node.Payload;
```



```
        InOrderTraversalIterator(node.Right, items, ref index);
     }
   }


  /************************************************************
   *  Pre-Order Traversal method
   ************************************************************/
   private void PostOrderTraversal(Node<TKey, TValue> node,
                                         KeyValuePair<TKey, TValue>[] items,
                                         ref int index){
     if(node != null){

        PostOrderTraversal(node.Right, items, ref index);
        items[index++] = node.Payload;
        PostOrderTraversal(node.Left, items, ref index);
     }
   }



   /*************************************************************
    *  ClearTree () method
    * ***********************************************************/
   private void ClearTree(Node<TKey, TValue> node){
     if(node != null){
       ClearTree(node.Left);
       ClearTree(node.Right);
       node.Left = null;
       node.Right = null;
       node.Payload = default(KeyValuePair<TKey, TValue>);

       _count--;
     }
   }


   /*************************************************************
    * PrintTreeToConsole()  method
    * ***********************************************************/
   public void PrintTreeToConsole() {
     foreach (KeyValuePair<TKey, TValue> item in this) {
       if (item.Key.CompareTo(_root.Payload.Key) == 0) {
         Console.ForegroundColor = ConsoleColor.Yellow;
         Console.Write(item.Key + " " + item.Value.ToString());
         Console.ForegroundColor = ConsoleColor.White;
       } else {
         Console.Write(item.Key + " " + item.Value.ToString());
       }
     }
     Console.WriteLine();
   }


   /*************************************************************
    *  PrintTreeStats()  method
    * ***********************************************************/
   public void PrintTreeStats() {
     Console.WriteLine("------------ Tree Stats --------------------");
     Console.WriteLine("First inserted key: {0}", _first_inserted_key);
     Console.WriteLine("Count: {0}", _count);
     Console.WriteLine("Left Rotates: {0}", _left_rotates);
     Console.WriteLine("Right Rotates: {0}", _right_rotates);
     Console.WriteLine("-------------------------------------------");
   }


    #region IXmlSerializable methods
   /********************************************************
     * WriteXml() method. Required by the IXmlSerializable interface
     ********************************************************/
   public void WriteXml(XmlWriter writer){
     XmlSerializer debugSerializer = new XmlSerializer(typeof(bool));
     XmlSerializer keySerializer = new XmlSerializer(typeof(TKey));
     XmlSerializer valueSerializer = new XmlSerializer(typeof(TValue));

     // write the state of the _debug field
     writer.WriteStartElement(DEBUG);
     debugSerializer.Serialize(writer, _debug);
     writer.WriteEndElement();

```



```
909     foreach(KeyValuePair<TKey, TValue> kvp in this.ToArray()){
910       writer.WriteStartElement(KEY_VALUE_PAIR);
911       writer.WriteStartElement(KEY);
912       keySerializer.Serialize(writer, kvp.Key);
913       writer.WriteEndElement();
914       writer.WriteStartElement(VALUE);
915       valueSerializer.Serialize(writer, kvp.Value);
916       writer.WriteEndElement();
917       writer.WriteEndElement();
918     }
919   }
920
921   /*******************************************************
922     * ReadXml() method - Required by IXmlSerializable interface
923     ********************************************************/
924   public void ReadXml(XmlReader reader){
925    XmlSerializer debugSerializer = new XmlSerializer(typeof(bool));
926    XmlSerializer keySerializer = new XmlSerializer(typeof(TKey));
927    XmlSerializer valueSerializer = new XmlSerializer(typeof(TValue));
928
929    bool emptyElement = reader.IsEmptyElement;
930    reader.Read();
931    if(emptyElement) return;
932
933     // read and set the state of the _debug field
934     reader.ReadStartElement(DEBUG);
935    _debug = (bool)debugSerializer.Deserialize(reader);
936     reader.ReadEndElement();
937
938    while(reader.NodeType != XmlNodeType.EndElement){
939      try{
940      reader.ReadStartElement(KEY_VALUE_PAIR);
941      reader.ReadStartElement(KEY);
942      TKey key = (TKey)keySerializer.Deserialize(reader);
943      reader.ReadEndElement();
944      reader.ReadStartElement(VALUE);
945      TValue value = (TValue)valueSerializer.Deserialize(reader);
946      reader.ReadEndElement();
947      this.Add(key,  value);
948      reader.ReadEndElement();
949      reader.MoveToContent();
950      }catch(Exception e){
951        Console.WriteLine(e);
952      }
953    }
954    reader.ReadEndElement();
955   }
956
957   /*******************************************************
958     * XmlSchema - Required by the IXmlSerializable interface
959     *                        returns null in this implementation
960     ********************************************************/
961   public XmlSchema GetSchema(){
962     return null;
963   }
964   #endregion
965
966
967   /*******************************************************
968      * NotifyCollectionChanged() method.
969      ********************************************************/
970   private void NotifyCollectionChanged(NotifyCollectionChangedEventArgs args){
971     if(CollectionChanged != null){
972       CollectionChanged(this, args);
973       if(_debug){
974         Console.WriteLine("------------Notify Collection Changed Event Raised----------------");
975       }
976     }
977
978   }
979   #endregion
980 } // end RedBlackTree class
```

Referring to example 18.13 — the RedBlackTree class has undergone extensive modification as a result of implementing the IDictionary<TKey, TValue>, INotifyCollectionChanged, and IXmlSerializable interfaces. Let's start by reviewing the changes made that are related to the IDictionary<TKey, TValue> interface.

The IDictionary<TKey, TValue> interface methods are gathered together in a #region beginning on line 134. The methods added include single and two-argument Add() methods, Clear(), Contains(), ContainsKey(), CopyTo(), single and two-argument Remove() methods, and the TryGetValue() method. I've also added a Keys property on line

123 that returns a collection of keys, and a Values property on line 127 that returns a collection of values. Let's look at the operation of the Add() methods. The Add() method on line 139 takes a populated KeyValuePair object as its argument and passes the call to the Insert() method. If the incoming item.Key or item.Value objects are null, the Insert() method throws an ArgumentNullException. The Add() method on line 146 takes two arguments, a key and a value, and passes the call onto the Insert() method. If the incoming key or value objects are null it throws an ArgumentNullException. I could rewrite this method in the fashion of the previous Add() method and let the Insert() method throw the ArgumentNullException. Following the call to the Insert() method note that I have modified it to raise the CollectionChanged event. This process occurs at two places in the method: first beginning on line 295 and the second time on line 342. In each case I first create a NotifyCollectionChangedEventArgs object and set its Action and NewItems properties via the constructor call. I then call the NotifyCollectionChanged() method which is defined on line 970.

Calls to either Remove() method pass the call on to the Delete() method. Removing an element from the tree also raises the CollectionChanged event. Note now, however, that the NotifyCollectionChangedEventArgs.Action property is set to NotifyCollectionChangedAction.Remove.

The RedBlackTree collection now contains two default enumerators: IEnumerable.GetEnumerator() on line 786 and GetEnumerator() on line 794. It also includes a named iterator HighToLow which begins on line 807. Note that the HighToLow iterator is a property, not a method. In summary, an enumerator method returns either an IEnumerator or an IEnumerator<T> (in this case an IEnumerator<KeyValuePair<TKey, TValue>>) while an iterator is a property that returns an IEnumerable <T> (in this case an IEnumerable<KeyValuePair<TKey, TValue>>).

The IEnumerable.GetEnumerator() method simply converts the tree to an array and returns the results of the GetEnumerator() call. On the other hand, the IEnumerable<T>.GetEnumerator() method, defined on line 794 traverses the tree with the InOrderTraversalIterator() method. The result of this traversal is an array which is then stepped through with a `for` statement on line 978 and each element is returned using the `yield` keyword.

The HighToLow iterator uses the PostOrderTraversal() method to return the tree's elements in descending order.

The IXmlSerializable interface methods begin on line 899 with the WriteXml() method. In the body of the WriteXml() method I create three XmlSerializers: one named debugSerializer to serialize the value of the _debug field, another named keySerializer to serialize the TKey type, and the last one named valueSerializer to serialize the TValue type. On line 905 I start the serialization of the RedBlackTree instance by writing a start element named Debug. I then serialize the value of the _debug field with a call to debugSerializer.Serialize(writer, _debug). I close the element tag with a call to writer.WriteEndElement(). Thus, if the `_debug` field is set to true, the previous three calls will result in the following XML element being serialized to an XML file: `<Debug>true</Debug>`

Following the serialization of the _debug field I serialize the key and value of each KeyValuePair in the body of the `foreach` loop that begins on line 909.

The ReadXml() method deserializes the elements of a RedBlackTree, starting with the value of the _debug field followed by the key and value of each KeyValuePair element. Note that the overall strategy of the ReadXml() method is to deserialize each KeyValuePair and then add the deserialized key and value to the tree with a call to the Add(key, value) method.

Let's look now at the modified RedBlackTree in action.

18.14 MainApp.cs

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Xml.Serialization;


public class MainApp {

 /******************************************************************
  * CollectionChangedEventHandler() method
  ****************************************************************/
  public static void CollectionChangedEventHandler(object sender, NotifyCollectionChangedEventArgs args){
  switch(args.Action){
    case NotifyCollectionChangedAction.Add :
       Console.WriteLine("CollectionChangedEvent Fired --> New object added to tree: " +
                         args.NewItems[0].ToString());
       break;
    case NotifyCollectionChangedAction.Replace :
       Console.WriteLine("CollectionChangedEvent Fired --> Object replaced --> Old Object " +
                         args.OldItems[0].ToString() + " New Object: " + args.NewItems[0]);
       break;
```



```
    case NotifyCollectionChangedAction.Remove :
       Console.WriteLine("CollectionChangedEvent Fired --> Object removed: " +
                          args.OldItems[0].ToString());
       break;
    case NotifyCollectionChangedAction.Reset :
       Console.WriteLine("CollectionChangedEvent Fired --> Collection cleared!");
       break;
  }
}

  /**********************************************************************
    * Main() method
    **********************************************************************/
  public static void Main(string[] args) {
    bool debugOn = false;
    if (args.Length > 0) {
      try {
        debugOn = Convert.ToBoolean(args[0]);
      } catch (Exception) {
        debugOn = false;
      }
    }

    RedBlackTree<PersonKey, Person> tree = new RedBlackTree<PersonKey, Person>(debugOn);
    tree.CollectionChanged += MainApp.CollectionChangedEventHandler;

    Person p1 = new Person("Deekster", "Willis", "Jaybones", Person.Sex.MALE, new DateTime(1966, 02, 19));
    Person p2 = new Person("Knut", "J", "Hampson", Person.Sex.MALE, new DateTime(1972, 04, 23));
    Person p3 = new Person("Katrina", "Kataline", "Kobashar", Person.Sex.FEMALE, new DateTime(1982, 09, 3));
    Person p4 = new Person("Dreya", "Babe", "Weber", Person.Sex.FEMALE, new DateTime(1978, 11, 25));
    Person p5 = new Person("Sam", "\"The Guitar Man\"", "Miller", Person.Sex.MALE,
                           new DateTime(1988, 04, 16));

    tree.Add(p1.Key, p1);
    tree.Add(p2.Key, p2);
    tree.Add(p3.Key, p3);
    tree.Add(p4.Key, p4);
    tree.Add(p5.Key, p5);

    tree.PrintTreeStats();
    Console.WriteLine("Original insertion order:");
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.WriteLine(p3);
    Console.WriteLine(p4);
    Console.WriteLine(p5);

    Console.WriteLine("------------------------------------------------");
    Console.WriteLine("Sorted Order:");
    tree.PrintTreeToConsole();


    /**********************************************
            Serialize Tree with XML Serializer
    **********************************************/
    TextWriter writer = null;
    FileStream fs = null;

    try{
      XmlSerializer serializer = new XmlSerializer(typeof(RedBlackTree<PersonKey, Person>));
      writer = new StreamWriter("datafile.xml");
      serializer.Serialize(writer, tree);
      writer.Close();


    }catch(Exception e){
      Console.WriteLine(e);
    }finally{
      if(writer != null){
        writer.Close();
      }
    }

     Console.WriteLine("----------- Deserializing Tree -----------");
    try{
      XmlSerializer serializer = new XmlSerializer(typeof(RedBlackTree<PersonKey, Person>));
      fs = new FileStream("datafile.xml", FileMode.Open);
      tree = null;
      tree = (RedBlackTree<PersonKey, Person>)serializer.Deserialize(fs);
      fs.Close();

```



```
104
105     }catch(Exception e){
106       Console.WriteLine(e);
107     }finally{
108       if(fs != null){
109         fs.Close();
110       }
111     }
112
113     Console.WriteLine("-----------Tree after XML deserialization -----------");
114     tree.PrintTreeToConsole();
115
116     // Reassign the event handler because we creamated the tree during XmlSerialization above...
117     tree.CollectionChanged += MainApp.CollectionChangedEventHandler;
118     Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
119                            new DateTime(1986, 02, 19));
120
121     tree[p6.Key] = p6;
122
123     tree.Remove(p4.Key);
124
125     Console.WriteLine("-----------Tree after modifications -----------");
126     tree.PrintTreeToConsole();
127
128     tree.Clear();
129
130   } // end Main() method
131 } // end MainApp class definition
```

Referring to example 18.14 — the MainApp class defines a method named CollectionChangedEventHandler() and a Main() method. In the body of the Main() method I check the value of the incoming command-line argument, the purpose of which is to set the value of the RedBlackTree's Debug property. If MainApp is executed without a command-line argument or the conversion throws an exception, the value of debugOn is set to false by default. An instance of RedBlackTree is created on line 46 followed by the assignment of the CollectionChangedEventHandler to its CollectionChanged event. Next, five Person objects are created and added to the tree. I then write the string representation of each Person object to the console to show the original insertion order, followed by a call to tree.PrintTreeToConsole() to show them in sorted order.

I then start the XML Serialization process on line 78. I serialize the tree using an XmlSerializer. I follow this with a deserialization process that begins on line 97. If all goes well the tree is serialized and deserialized. The result of this program is a play-by-play console display and an XML file named DataFile.xml which I've listed in example 18.15. Figure 18-5 shows the results of running this program.

18.15 DataFile.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RedBlackTreeOfPersonKeyPerson>
  <Debug>
    <boolean>false</boolean>
  </Debug>
  <KeyValuePair>
    <Key>
      <PersonKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
        <KeyString>Deekster Willis Jaybones  MALE  44 d95196ed-4b65-4c8c-b158-1594d306e4e8</KeyString>
      </PersonKey>
    </Key>
    <Value>
      <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
        <FirstName>Deekster</FirstName>
        <MiddleName>Willis</MiddleName>
        <LastName>Jaybones</LastName>
        <Gender>MALE</Gender>
        <Birthday>1966-02-19T00:00:00</Birthday>
        <DNA>d95196ed-4b65-4c8c-b158-1594d306e4e8</DNA>
      </Person>
    </Value>
  </KeyValuePair>
  <KeyValuePair>
    <Key>
      <PersonKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
        <KeyString>Dreya Babe Weber  FEMALE  31 59631071-df1a-4911-adf6-777c6a6d951f</KeyString>
      </PersonKey>
    </Key>
    <Value>
```

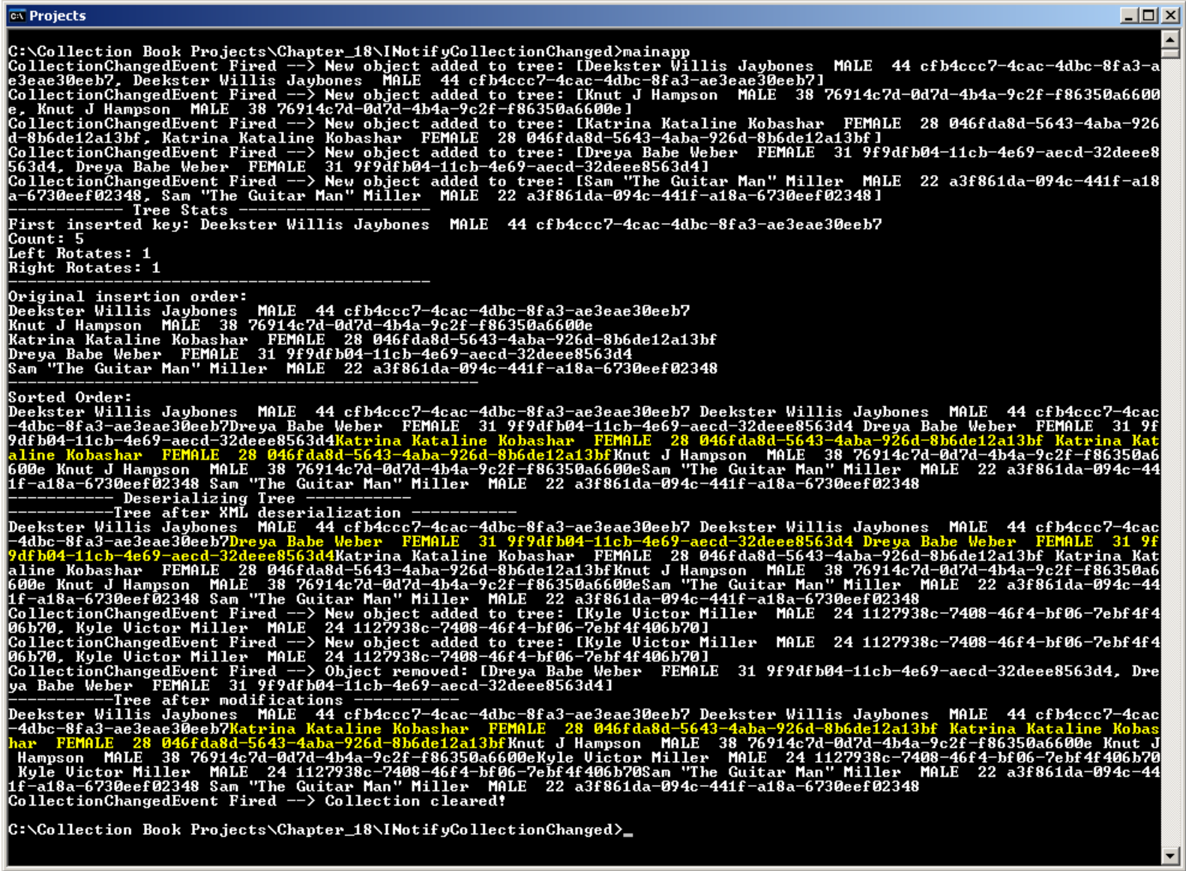



Figure 18-5: Results of Running Example 18.14

```
          <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <FirstName>Dreya</FirstName>
            <MiddleName>Babe</MiddleName>
            <LastName>Weber</LastName>
            <Gender>FEMALE</Gender>
            <Birthday>1978-11-25T00:00:00</Birthday>
            <DNA>59631071-df1a-4911-adf6-777c6a6d951f</DNA>
          </Person>
        </Value>
      </KeyValuePair>
      <KeyValuePair>
        <Key>
          <PersonKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <KeyString>Katrina Kataline Kobashar  FEMALE  28 574a3049-94bf-46a8-876c-9c1c47dc56b8</KeyString>
          </PersonKey>
        </Key>
        <Value>
          <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <FirstName>Katrina</FirstName>
            <MiddleName>Kataline</MiddleName>
            <LastName>Kobashar</LastName>
            <Gender>FEMALE</Gender>
            <Birthday>1982-09-03T00:00:00</Birthday>
            <DNA>574a3049-94bf-46a8-876c-9c1c47dc56b8</DNA>
          </Person>
        </Value>
      </KeyValuePair>
      <KeyValuePair>
        <Key>
          <PersonKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <KeyString>Knut J Hampson  MALE  38 058e82b9-7ac6-4ca4-b9b4-beb471b1d21d</KeyString>
          </PersonKey>
```



```
        </Key>
        <Value>
          <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <FirstName>Knut</FirstName>
            <MiddleName>J</MiddleName>
            <LastName>Hampson</LastName>
            <Gender>MALE</Gender>
            <Birthday>1972-04-23T00:00:00</Birthday>
            <DNA>058e82b9-7ac6-4ca4-b9b4-beb471b1d21d</DNA>
          </Person>
        </Value>
      </KeyValuePair>
      <KeyValuePair>
        <Key>
          <PersonKey xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <KeyString>Sam "The Guitar Man" Miller  MALE  22 73a4f3d9-fbb6-4874-a06f-f584b0db523e</KeyString>
          </PersonKey>
        </Key>
        <Value>
          <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
            <FirstName>Sam</FirstName>
            <MiddleName>"The Guitar Man"</MiddleName>
            <LastName>Miller</LastName>
            <Gender>MALE</Gender>
            <Birthday>1988-04-16T00:00:00</Birthday>
            <DNA>73a4f3d9-fbb6-4874-a06f-f584b0db523e</DNA>
          </Person>
        </Value>
      </KeyValuePair>
    </RedBlackTreeOfPersonKeyPerson>
```


Quick Review

If your custom collection conforms to the behavior of an existing .NET collection framework interface then simply implement the interface to make your collection conformant. Implement additional interfaces as necessary to support custom serialization or event notification.

The IEnumerable.GetEnumerator() and IEnumerable<T>.GetEnumerator() methods represent default enumerators. These methods usually step through a collection in ascending order using the `foreach` statement. Supply a named iterator if you want to provide an alternative enumeration for a collection.

Custom Collection Implementation Summary Table

Table 18-1 summarizes custom collection implementation considerations and courses of action.

Consideration	Course of Action
Should you extend an existing collection or create one from scratch?	If the base collection provides a majority of the functionality you require then extending the collection and overriding selected methods or adding new ones as appropriate will save you time and money.
Does your collection exhibit the behavior of an existing .NET collections framework interface?	If so, implement the interface taking care to study the expected behavior of each interface method and expected exceptions should something go wrong.
Do you want to interate over the elements of your collection with the foreach statement?	If so, implement the IEnumerable and IEnumerable<T> interfaces. You may also need to create a custom enumerator by implementing the IEnumerator and IEnumerator<T> interfaces.

Table 18-1: Custom Collection Implementation Summary Table

Consideration	Course of Action
Implementing IEnumerable.GetEnumerator()	The IEnumerable.GetEnumerator() method is the non-generic default enumerator. It returns an IEnumerator. Note that to differentiate this version of the GetEnumerator() method you must fully qualify it by appending the interface name to the method name using the dot '.' operator like so: IEnumerable.GetEnumerator().
Implementing IEnumerable<T>.GetEnumerator()	This is the generic default enumerator. It returns IEnumerator<T>. Use the yield keyword to return individual collection elements. The use of the yield keyword hides the complexities ordinarily associated with implementing enumerators in earlier version of the .NET Framework.
Named Iterators	A named iterator is a readonly property that returns an IEnumerable<T> object. Implement a named iterator if you need to provide an alternative ordering for your collection.

Table 18-1: Custom Collection Implementation Summary Table

Summary

You can create a custom collection by extending an existing collection and providing your own implementations of the required class members. The need to create custom, non-generic, strongly-typed collection classes has been rendered an obsolete practice with the introduction of generics. However, not all systems running the .NET framework can update to the latest framework release, so you may encounter legacy code running on production systems that utilize custom, non-generic collections.

If your custom collection conforms to the behavior of an existing .NET collection framework interface then implement the interface. Implement additional interfaces as necessary to support custom serialization or event notification.

The IEnumerable.GetEnumerator() and IEnumerable<T>.GetEnumerator() methods represent default enumerators. These methods usually step through a collection in ascending order using the `foreach` statement. Supply a named iterator if you want to provide an alternative enumeration for a collection.

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

Notes

Chapter 19


Leica MP

Sport's Shoes

Specialized Collections

Learning Objectives

- List and describe the members of the System.Collections.Specialized namespace
- List the specialized classes that have been rendered obsolete by generics
- Describe the unique characteristics of a HybridDictionary
- Describe how the indexer is overloaded in the NameValueCollection
- List several uses for the BitVector32 class
- Describe the problem solved by the Weak Event Listener pattern
- Implement the IWeakEventListener interface
- Use the NameValueCollection class in a program
- Use the HybridDictionary class in a program
- Use the BitVector32 class in a program
- Use the CollectionChangedEventManager class in a program

Introduction

The System.Specialized namespace, at its name implies, is where you'd look for collection classes that offer unique or specialized behavior. At first glance it may seem like most of these classes, especially ones that have been around since .NET framework version 1.0, would have been rendered obsolete with the introduction of generics, but a deeper inspection reveals this has happened to only one or two. A few of the classes in the System.Specialized namespace truly do have special powers.

In this chapter I'm only going to cover a handful of System.Specialized namespace members. These include the NameValueCollection, the HybridDictionary, the BitVector32, and the CollectionChangedEventManager and its related IWeakEventListener interface.

The NameValueCollection is one of those classes that at first glance looks like it could be replaced by an existing generic class. If it behaved as a straightforward dictionary your could do that, but elements within the NameValueC-ollection can also be accessed via a numeric index.

The HybridDictionary has dual personalities: It starts of internally as a linked list, but switches to a hash table when the number of elements it contains exceeds a certain threshold.

The BitVector32 structure allows you to store multiple bit or several small integer values in the space of one 32-bit data structure.

The CollectionChangedEventManager is an implementation of the weak event listener pattern. I'll show you how to implement the IWeakEventListener interface on an event listener object and then use it to respond to CollectionChangedEvents.

System.Collections.Specialized Namespace

Most of the members of the System.Specialized namespace have been around since version 1.0 of the .NET framework with relatively few additions since. Table 19-1 lists the namespace members and describes what they do.

Namespace Member Name	Since .NET Version	Comments
Classes		
CollectionChangedEventManager	3.0	An implementation of System.Windows.WeakEventManager. Allows you to use the *weak event listener pattern* to respond to CollectionChanged events.
CollectionsUtil	1.0	Lets you create case-insensitive collections of various types. Case-insensitive collections ignore character case when making string comparisons.
HybridDictionary	1.0	An interesting class that stores small numbers of elements in a linked list (ListDictionary), but changes internally to a hash table (System.Collections.HashTable) when the number of elements exceeds a threshold.
ListDictionary	1.0	A collection whose underlying data structure is a linked list. Efficient when elements number less than 10.
NameObjectCollectionBase	1.0	Abstract base class for collections consisting of key/value pairs where the key is a string and values are objects.

Table 19-1: System.Collections.Specialized Namespace Members

Namespace Member Name	Since .NET Version	Comments
NameObjectCollectionBase.Keys-Collection	1.0	A strongly-typed collection of key strings.
NameValueCollection	1.0	A strongly-typed collection of key/value pairs where the keys are strings and the values are strings. Items in a NameValueCollection can be accessed either via their key or by their index.
NotifyCollectionChangedEvent-Args	3.0	Used to pass data related to a CollectionChanged event.
OrderedDictionary	2.0	A collection of key/value pairs accessible by key or index.
StringCollection	1.0	A strongly-typed collection of string objects. Use List<String> instead.
StringDictionary	1.0	A strongly-typed hashtable of string keys and string values. The difference between StringDictionary and NameValueCollection is that items in a NameValueCollection can be accessed via an index in addition to its key.
StringEnumerator	1.0	Implements an iteration over the items in a StringCollection.
Structures		
BitVector32	1.0	Used to store Boolean values and multiple small integers in 32 bits of memory.
BitVector32.Section	1.0	Represents a particular section within a BitVector32 data structure. Properties include Mask and Offset.
Interfaces		
INotifyCollectionChanged	3.0	Enables collections to raise the CollectionChanged event in response to items being added, updated, or removed from the collection.
IOrderedDictionary	2.0	Represents a dictionary whose items can be accessed via an index or via a key.
Delegates		
NotifyCollectionChangedEvent-Handler	3.0	Specifies the signature of methods that respond to CollectionChanged events. It's also the delegate type of the CollectionChanged event.
Enumerations		
NotifyCollectionChangedAction	3.0	Represents the various actions related to a CollectionChanged event. Conveyed as the Action property of the NotifyCollectionChangedEventArgs class. Actions include Add, Remove, Replace, Move, and Reset.

Table 19-1: System.Collections.Specialized Namespace Members

Referring to table 19-1 — you've already seen the INotifyCollectionChanged interface used earlier in the book in chapter 18 when I transformed the RedBlackTree into a full-fledged collection. The INotifyCollectionChanged interface is closely related to the CollectionChangedEventManager as you'll see later.

The only specialized collection that can be replaced outright by an existing generic class is the StringCollection. It can be replaced by a generic List<String>.

The BitVector32 is an interesting collection that can store and manipulate multiple bit values or multiple small integral values consisting of 2 or more bits. Its one of those classes that are difficult to use unless you see good example of it in action.

Quick Review

The System.Collections.Specialized namespace contains collection classes and other supporting members with unique capabilities. Most of the members of the System.Collections.Specialized namespace have been around since the .NET framework version 1.0 and while one would think that generics would render most of them obsolete, that's not necessarily the case. Only the StringCollection can be replaced outright by a generic List<String>.

NameValueCollection

The NameValueCollection class is a cross between a dictionary and a list. By this I mean you can insert key/value pairs and access each value via its key or via an integer indexer. Figure 19-1 shows the UML class diagram for the NameValueCollection inheritance hierarchy.

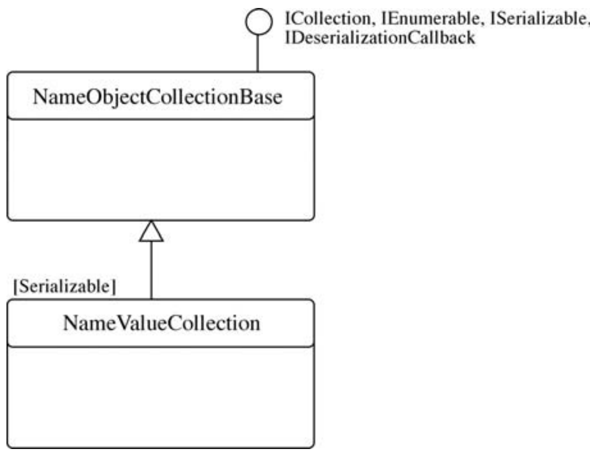


Figure 19-1: NameValueCollection Inheritance Diagram

Referring to figure 19-1 — the NameValueCollection extends the NameObjectCollectionBase class. The NameValueCollection is tagged with the [Serializabe] attribute and its elements can be iterated with a `foreach` statement.

Example 19.1 demonstrates the use of the NameValueCollection in a short program that scans a sequence of IP addresses and looks up their corresponding DNS hostnames. In this example I've kept the range of IP addresses narrow to keep the running time of the program reasonable.

19.1 NameValueCollectionDemo.cs

```
using System;
using System.Collections.Specialized;
using System.Net;
using System.IO;

public class NameValueCollectionDemo {

   public static void Main(){
     NameValueCollection nvc = new NameValueCollection();
    FileStream fs = null;
    StreamWriter writer = null;

    for(int i = 122; i<123; i++){
     for(int j = 183; j<184; j++){
      for(int k = 97; k<99; k++){
       for(int l = 50; l<88; l++){
          try{
          String ipaddress_string = i + "." + j + "." + k + "." + l;
```



```
          IPAddress ipaddress = IPAddress.Parse(ipaddress_string);
          string hostname = Dns.GetHostEntry(ipaddress).HostName;
          Console.WriteLine(hostname + " : " + ipaddress_string);
          nvc.Add(hostname, ipaddress_string);
          }catch(Exception){
             // ignored
          }
       }
     }
    }
 }  // end outer for loop

     try{
       fs = new FileStream("hostnames.txt", FileMode.Create);
       writer = new StreamWriter(fs);
       foreach(string hostname in nvc.Keys){
         writer.Write(hostname + " | " + nvc[hostname] + "\r\n");
     }
     writer.Flush();

    }catch(Exception e){
      Console.WriteLine(e);
    }finally{
      try{
       if(writer != null){
          writer.Close();
          fs.Close();
       }
     }catch(Exception){
        //ignored
     }
    }

  } // end Main()
}
```

Referring to example 19.1 — the IP addresses scanned in this program range from 122.183.97.50 - 122.183.98.87. If an IP address has a DNS host entry I add the hostname and the IP address to an instance of NameValueCollection called nvc. When the scan completes I write each hostname and IP address to a text file named hostnames.txt. Figure 19-2 shows the partial results of running this program.

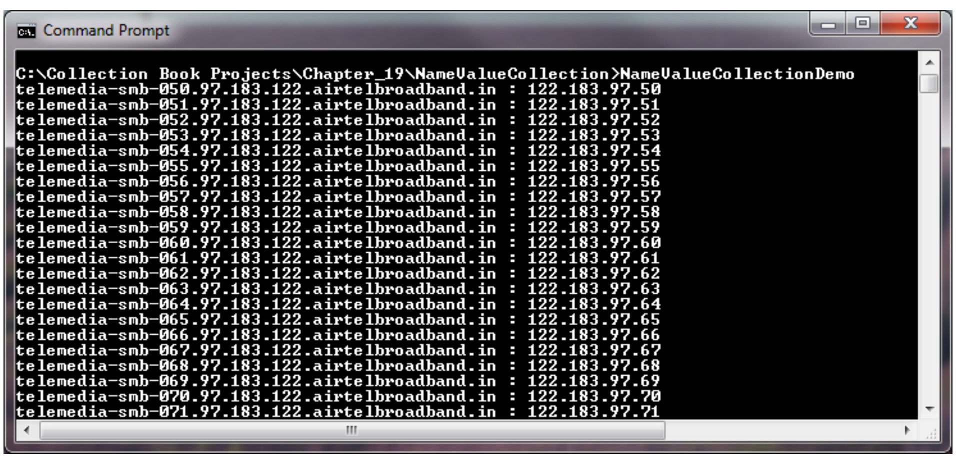


Figure 19-2: Results of Running Example 19.1

Quick Review

The NameValueCollection is a cross between a dictionary and a list. It supports the access of individual elements via a key or via a numeric indexer.

HybridDictionary

The HybridDictionary is a cross between a ListDictionary and a System.Collections.HashTable. It starts off by storing elements in a linked list (ListDictionary). When the number of elements exceeds a set threshold it switches internally to a hash table (HashTable).

Figure 19-3 gives the UML class diagram for the HybridDictionary.

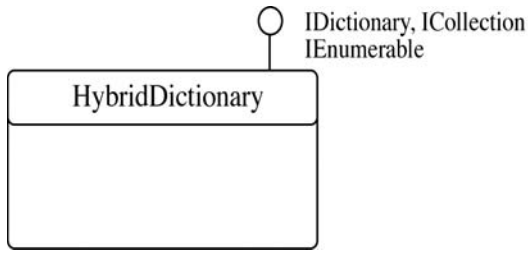


Figure 19-3: HybridDictionary Class Diagram

Referring to figure 19-3 — the HybridDictionary implements the IDictionary, ICollection, and IEnumerable interfaces. Its elements can be accessed by key and they can be iterated over with a `foreach` statement.

Example 19.2 offers a short program showing the HybridDictionary in action. In this simple example I offer some advice to myself and a few of my friends by way of their horoscopes.

19.2 HybridDictionaryDemo.cs

```
1   using System;
2
3
4   using System.Collections.Specialized;
5
6
7   public class HybridDictionaryDemo  {
8     public static void Main(){
9       HybridDictionary hd = new HybridDictionary();
10      hd.Add("Rick", "Aquarius: The time is right to travel! Stay receptive, " +
11          "new opportunities will present themselves.");
12      hd.Add("Coralie", "Aries: Move forward with your big plans! The time is right to stike.");
13      hd.Add("Kyle", "Leo: Your mind's made up! Don't procrastinate.");
14      hd.Add("Tati", "Sagittarius: Finish college before it's too late!");
15      hd.Add("Sport", "Gemini: Take charge! Ask him to marry you!");
16      hd.Add("John", "Gemini: Surrender yourself! Say yes when a beautiful woman asks you to marry her!");
17
18      Console.WriteLine("Name \t\t Horoscope");
19      Console.WriteLine("------------------------");
20
21
22      foreach(string s in hd.Keys){
23        Console.WriteLine(s + "\t\t" + hd[ s] );
24      }
25
26    } // end Main()
27  } // end class definition
```

Referring to example 19.2 — on line 9 I create an instance of HybridDictionary using the default constructor. On lines 10 through 16 I add horoscope entries in the form of key/value pairs using the Add() method. On lines 18 and 19 I set up the column headings and the `foreach` statement on line 22 looks up each value by key and prints the resulting key and value to the console. Figure 19-4 shows the results of running this program.

Quick Review

The HybridDictionary is a cross between a ListDictionary and a System.Collections.HashTable. It starts off by storing elements in a linked list (ListDictionary). When the number of elements exceeds a set threshold it switches internally to a hash table (HashTable).

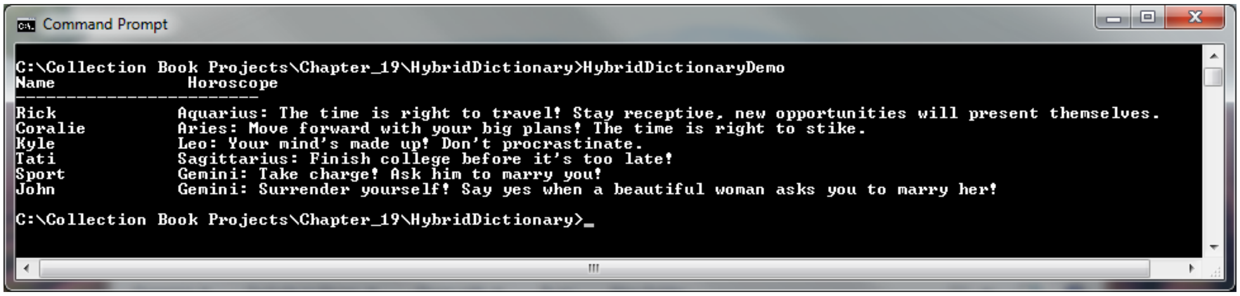


Figure 19-4: Results of Running Example 19.2

BitVector32

The BitVector32 structure allows you to store individual bits or larger integral values in one 32-bit structure. Figure 19-5 gives the UML class diagram for a BitVector32.

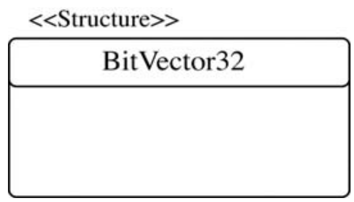


Figure 19-5: BitVector32 Class Diagram

To manipulate individual bits in the BitVector32 structure you must first create masks via the static BitVector32.CreateMask() method. Once you've created a mask, you use the mask along with an indexer to set the bit represented by that mask to either true or false.

Alternatively, you can divvy up the BitVector32 into sections using the static BitVector32.CreateSection() method. Each section, represented by the BitVector32.Section structure, contains the minimum number of bits required to represent the integer value supplied to the CreateSection() method.

Example 19.3 shows how to store individual bit values using masks and larger integral values using sections.

19.3 BitVector32Demo.cs

```
using System;
using System.Collections.Specialized;

public class BitVector32Demo {
  public static void Main(){
    // Store and access individual bit values using masks
    BitVector32 bv1 = new BitVector32(0);
    Console.WriteLine("bv1 intial value: \t" + (Convert.ToString(bv1.Data, 2)).PadLeft(32, '0'));

    int bit1 = BitVector32.CreateMask();
    int bit2 = BitVector32.CreateMask(bit1);
    int bit3 = BitVector32.CreateMask(bit2);

    bv1[bit1] = true;
    bv1[bit3] = true;

    Console.WriteLine("bv1 b1 and b3 true: \t" + (Convert.ToString(bv1.Data, 2)).PadLeft(32, '0'));
    Console.WriteLine("--------------------------------------------------------");

    // Store and access larger values using sections

    BitVector32 bv2 = new BitVector32(0);
    Console.WriteLine("bv2 intial value: \t" + (Convert.ToString(bv2.Data, 2)).PadLeft(32, '0'));

    BitVector32.Section s1 = BitVector32.CreateSection(8); // uses first 4 bits
    BitVector32.Section s2 = BitVector32.CreateSection(4, s1); // uses next 3 bits following s1

    bv2[s1] = 6;
```



```
29      bv2[ s2]  = 4;
30
31      Console.WriteLine("bv2 s1 and s2: \t\t" + (Convert.ToString(bv2.Data, 2)).PadLeft(32, '0'));
32    } // end Main()
33  } // end class
```

Referring to example 19.3 — the first BitVector32 instance named bv1 is used to store individual bit values. The instance is created on line 7 and line 8 prints the initial state of 32 bits to the console. Lines 10 through 12 create bit masks named bit1, bit2, and bit3. Note how the first bit mask is created with the no-argument version of the CreateMask() method but each subsequent mask is created with the help of the previous mask. To set the bit represented by its mask, use the indexer along with the bit mask as shown on lines 14 and 15. In this case I'm using the bit1 and bit3 masks to set those bits to true. Line 17 prints the state of bv1 to the console.

Sections are similar to masks but encompass more than one bit. The number of bits reserved for each section is set with the CreateSection() method. On lines 25 and 26 I create two sections named s1 and s2. Section s1 contains the minimum number of bits (4) required to represent numbers between 0 and 8. Section s2 uses 3 bits to represent numbers between 0 and 4. Note on line 26 how section s1 is used to define section s2. Lines 28 and 29 show how to access each section via an indexer and the section name. Figure 19-6 shows the results of running this program.

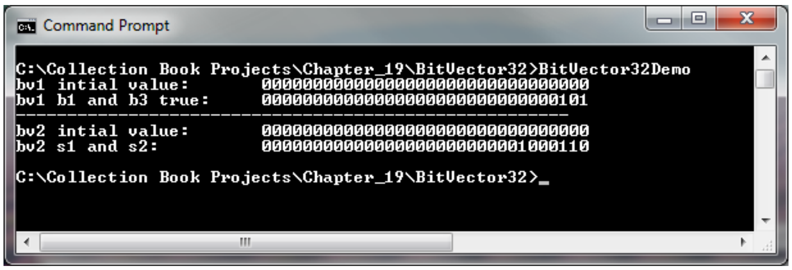


Figure 19-6: Results of Running Example 19.3

Quick Review

The BitVector32 structure allows you to store individual bits or larger integral values in one 32-bit structure. To manipulate individual bits in the BitVector32 structure you must first create masks via the static BitVector32.CreateMask() method. Once you've created a mask, you use the mask along with an indexer to set the bit represented by that mask to either true or false. Alternatively, you can divvy up the BitVector32 into sections using the static BitVector32.CreateSection() method. Each section, represented by the BitVector32.Section structure, contains the minimum number of bits required to represent the integer value supplied to the CreateSection() method.

CollectionChangedEventManager

The CollectionChangedEventManager is a key player in the weak event listener pattern as applied to CollectionChangedEvents.

The weak event listener pattern attempts to solve a potential memory leak problem that can arise when event listeners are added to events using the '+=' operator.

Normally, event handler methods are added to an event using the '+=' operator in the following fashion:

```
source_object.Event += listener_object.EventHandlerMethod;
```

This is further illustrated in figure 19-7.

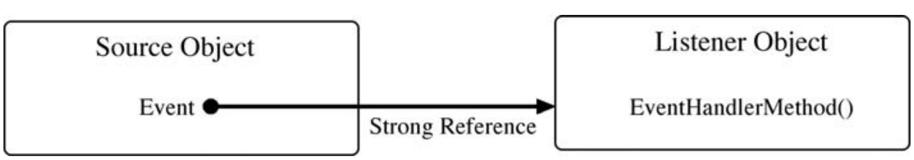


Figure 19-7: Strong Reference between a Source Object Event and Listener Object that Contains the EventHandlerMethod()

Referring to figure 19-7 — a strong reference exists between the source_object.Event and the listener_object that contains the EventHandlerMethod(). In the event that the listener_object might go out of scope before the source_object does, the garbage collector will not reclaim the listener object while the source_object.Event retains a strong reference to it in its list of event handlers.

The weak event listener pattern breaks this strong reference relationship between source objects and listener objects by inserting an intermediary or manager object between the source and listener objects. This pattern is illustrated in figure 19-8.

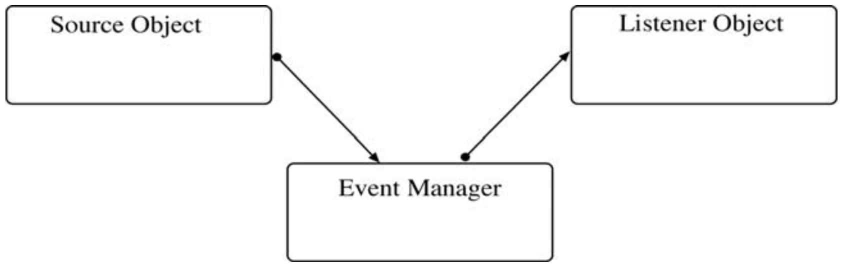


Figure 19-8: Weak Event Listener Pattern

Referring to figure 19-8 — the event manager decouples the source object from the listener object, making it easier for the garbage collector to reclaim the listener object should its lifetime end before the source object's.

Event managers are created to handle specific types of events. Listener objects implement the System.Windows.IWeakEventListener interface which specifies one method named ReceiveWeakEvent().

The CollectionChangedEventManager is a WeakEventManager designed specifically to manage the connections between collections that implement the INotifyCollectionChanged interface and listeners that implement the IWeakEventListener interface. Figure 19-9 shows the UML class diagram for the CollectionChangedEventManager.

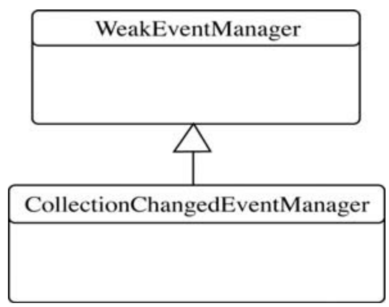


Figure 19-9: CollectionChangedEventManager Inheritance Class Diagram

To demonstrate the use of the CollectionChangedEventManager I'll use the RedBlackTree collection developed in chapter 18. The RedBlackTree class implements the INotifyCollectionChanged interface which specifies the CollectionChanged event.

You can compile the RedBlackTree class, along with its related Node class, into a dll using the following compiler command:

```
csc /t:library RedBlackTree.cs Node.CS
```

This results in a dll named RedBlackTree.dll.

You'll also need the Person class, which you can obtain from the same chapter examples. Compile the Person class into a dll using the following compiler command:

```
csc /t:library Person.cs
```

This results in a dll named Person.dll.

Next, and this is quite important, you'll need to search for a library named WindowsBase.dll which should be located in your .NET framework installation directory. On my machine the file resides on the following absolute path:

```
c:\Windows\Microsoft.NET\Framework64\v4.0.30319\WPF\WindowsBase.dll
```

Copy the WindowsBase.dll to the project folder you're using for this example.

Example 19.4 gives the code for a class named ListenerObject. The ListenerObject class implements the IWeakEventListener interface.

19.4 ListenerObject.cs

```
1  using System;
2  using System.Text;
3  using System.Windows;
4  using System.Collections.Specialized;
5  using System.Net.Mail;
6
7  public class ListenerObject : IWeakEventListener {
8
9     public bool ReceiveWeakEvent(Type managerType, object sender, EventArgs e){
10      if(managerType == typeof(CollectionChangedEventManager)){
11       CollectionChangedHandler(sender, (NotifyCollectionChangedEventArgs)e);
12       return true;
13   }
14   return false;
15    }
16
17    private void CollectionChangedHandler(object sender, NotifyCollectionChangedEventArgs e){
18      try{
19       SmtpClient smtp_client = new SmtpClient("127.0.0.1", 25);
20       MailAddress from = new MailAddress("rick@pulpfreepress.com");
21       MailAddress to = new MailAddress("rick@pulpfreepress.com");
22       MailMessage email = new MailMessage(from, to);
23       email.Subject = sender.ToString() + " has changed...";
24       email.Body = "Tree has changed!";
25       smtp_client.Send(email);
26       Console.WriteLine("CollectionChangedEvent fired! Email notification sent to {0}!", to.ToString());
27     }catch(Exception ex){
28       Console.WriteLine("Collection has changed, email notification not sent. Check SMTP settings...");
29       Console.WriteLine(ex);
30     }
31    }
32
33 }
```

Referring to example 19.4 — the ListenerObject implements the IWeakEventListener interface which specifies one method named ReceiveWeakEvent(). The ReceiveWeakEvent() method is automatically called by an event manager which passes in the three required arguments. Your job as implementor of the ReceiveWeakEvent() method is to figure out what type of event manager called the method and respond accordingly. This occurs in the `if` statement which begins on line 10. I pass the processing on to a method named CollectionChangedHandler(), which attempts to send an email notification. If the email notification fails for any reason and throws an exception, I simply write a brief message to the console.

To get the email notification code to work you'll need to modify the values of the SmtpClient IP address and port number and set the to and from email addresses accordingly. I don't mind getting emails from readers, but I'd rather not receive an email every time you update your red black tree!

Example 19.5 gives the code for a short program that demonstrates the use of the CollectionChangedEventManager.

19.5 MainApp.cs

```
using System;
using System.Collections.Specialized;

public class MainApp {

  [STAThread]
  public static void Main(string[] args) {
    bool debugOn = false;
    if (args.Length > 0) {
      try {
        debugOn = Convert.ToBoolean(args[0]);
      } catch (Exception) {
        debugOn = false;
      }
    }

    RedBlackTree<PersonKey, Person> tree = new RedBlackTree<PersonKey, Person>(debugOn);
    ListenerObject listener = new ListenerObject();
    CollectionChangedEventManager.AddListener(tree, listener);

    Person p1 = new Person("Deekster", "Willis", "Jaybones", Person.Sex.MALE, new DateTime(1966, 02, 19));
    Person p2 = new Person("Knut", "J", "Hampson", Person.Sex.MALE, new DateTime(1972, 04, 23));
    Person p3 = new Person("Katrina", "Kataline", "Kobashar", Person.Sex.FEMALE, new DateTime(1982, 09, 3));
     Person p4 = new Person("Dreya", "Babe", "Weber", Person.Sex.FEMALE, new DateTime(1978, 11, 25));
     Person p5 = new Person("Sam", "\"The Guitar Man\"", "Miller", Person.Sex.MALE,
                            new DateTime(1988, 04, 16));

```



```
28      try{
29      tree.Add(p1.Key, p1);
30      tree.Add(p2.Key, p2);
31      tree.Add(p3.Key, p3);
32      tree.Add(p4.Key, p4);
33      tree.Add(p5.Key, p5);
34      }catch(Exception ex){
35         Console.WriteLine(ex);
36      }
37
38      tree.PrintTreeStats();
39      Console.WriteLine("Original insertion order:");
40      Console.WriteLine(p1);
41      Console.WriteLine(p2);
42      Console.WriteLine(p3);
43      Console.WriteLine(p4);
44      Console.WriteLine(p5);
45
46      Console.WriteLine("-------------------------------------------------");
47      Console.WriteLine("Sorted Order:");
48      tree.PrintTreeToConsole();
49
50    }  // end Main()
51
52
53  } // end MainApp class
```

Referring to example 19.5 — on line 17 I create the RedBlackTree object as usual. On line 18 I create an instance of the ListenerObject named simply listener. On line 19 I use the CollectionChangedEventManager's static AddListener() method to connect the tree and the listener. The rest of this program remains unchanged from the original version developed in chapter 18.

Now, to compile this example using the command line compiler, it's easier if you've placed all the required files in one folder. Figure 19-10 shows the list of files in my project folder. In addition to ListenerObject.cs, MainApp.cs, Person.dll, RedBlackTree.dll, and WindowsBase.dll, I have added a file named compile.bat that contains the compilation command as shown in example 19.6

19.6 compile.bat

```
csc /r:RedBlackTree.dll;Person.dll;WindowsBase.dll *.cs
```

Referring to example 19.6 — this command references the required dlls and compiles the two source files ListenerObject.cs and MainApp.cs. Figure 19-11 shows the results of running example 19.5.

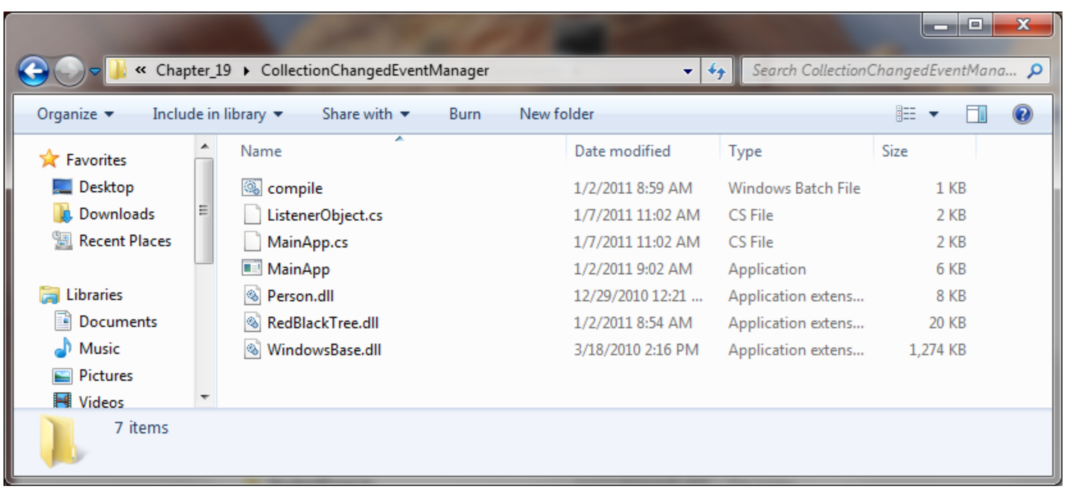


Figure 19-10: All Required Files In One Project Folder

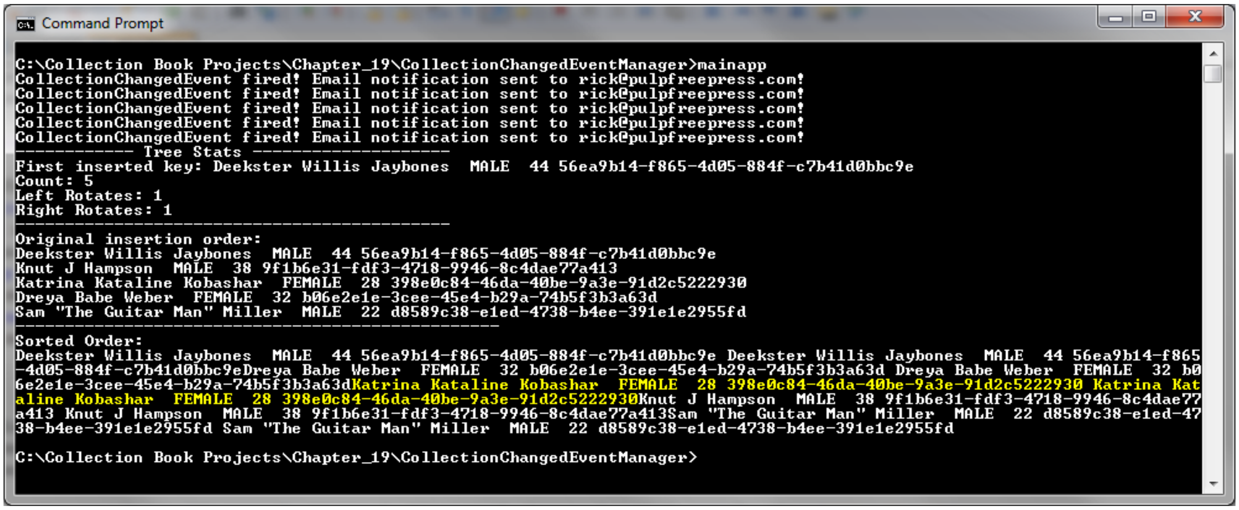


Figure 19-11: Results of Running Example 19.5

Quick Review

The CollectionChangedEventManager is a key player in the weak event listener pattern as applied to Collection-ChangedEvents. The weak event listener pattern attempts to solve a potential memory leak problem that can arise when event listeners are added to events using the '+=' operator. The weak event listener pattern breaks this strong reference relationship between source objects and listener objects by inserting an intermediary or manager object between the source and listener objects.

Summary

The System.Collections.Specialized namespace contains collection classes and other supporting members with unique capabilities. Most of the members of the System.Collections.Specialized namespace have been around since the .NET framework version 1.0 and while one would think that generics would render most of them obsolete, that's not necessarily the case. Only the StringCollection can be replaced outright by a generic List<String>.

The NameValueCollection is a cross between a dictionary and a list. It supports the access of individual elements via a key or via a numeric indexer.

The HybridDictionary is a cross between a ListDictionary and a System.Collections.HashTable. It starts off by storing elements in a linked list (ListDictionary). When the number of elements exceeds a set threshold it switches internally to a hash table (HashTable).

The BitVector32 structure allows you to store individual bits or larger integral values in one 32-bit structure. To manipulate individual bits in the BitVector32 structure you must first create masks via the static BitVector32.Create-Mask() method. Once you've created a mask, you use the mask along with an indexer to set the bit represented by that mask to either true or false. Alternatively, you can divvy up the BitVector32 into sections using the static BitVector32.CreateSection() method. Each section, represented by the BitVector32.Section structure, contains the minimum number of bits required to represent the integer value supplied to the CreateSection() method.

The CollectionChangedEventManager is a key player in the weak event listener pattern as applied to Collection-ChangedEvents. The weak event listener pattern attempts to solve a potential memory leak problem that can arise when event listeners are added to events using the '+=' operator. The weak event listener pattern breaks this strong reference relationship between source objects and listener objects by inserting an intermediary or manager object between the source and listener objects.

References

Microsoft Developer Network (MSDN) .NET Framework 4.0 Documentation [http://www.msdn.com]

Notes

Chapter 20

Leica MP

Once Upon a Time

Collections In Action

Learning Objectives

- *Demonstrate your ability to utilize collections in your class design*
- *Use a List<T> to represent a one-to-many relationship between entities*
- *Design and build a multitiered, networked, data-driven, client-server application*
- *Use Structured Query Language (SQL) to manipulate a relational database*
- *Define the terms "table", "row", "column", "primary key", "foreign key", and "constraint"*
- *Use data access objects (DAOs) to map objects to relational database tables*
- *Use business objects (BOs) to implement business logic*
- *Use value objects (VOs) to model application entities and transfer data between tiers*
- *Use a Microsoft Enterprise Library Data Access Block to build a data-driven, client-server application*
- *Use the DatabaseFactory class to create a database connection*
- *Use prepared statements to execute SQL commands*
- *Use prepared statement parameters to build dynamic SQL commands*
- *Correlate a C# data type to its corresponding Microsoft SQL Server data type*
- *Manipulate large binary database objects*
- *Use a DataGridView to display and manipulate tabular data in a graphical user interface (GUI)*

Introduction

As the title I've given this chapter implies, I want to demonstrate collections in action in a real-world scenario, however, as you certainly have guessed from reading the learning objectives, this chapter covers much more than that. It shows you how to build a multitiered, networked, client-server, database application using Data Access Objects (DAOs), Business Objects (BOs), and Value Objects (VOs).

Your success in completing this chapter hinges on your ability to properly install and configure several critical components. These include Microsoft SQLServer 2008 R2 and Microsoft Enterprise Library version 5.0. The installation of both of these tools is relatively painless and straightforward, however, as with most things in the programming world, you may have to fiddle with things to get them to work.

As you know by now, programming, in large part, requires relentless attention to detail. At no other time is this more true than when you start adding the complexities of database access to the mix. One small spelling mistake in a configuration file or SQL query will render an application inoperable. Also, at the start, you may feel overwhelmed by the myriad complexities that confront you. There's the database, SQL syntax, relational database theory, new terms and technology, and the complexity of a multitiered application. To get a complex application to run correctly requires each piece of the application to work correctly. But fear not. At every step of the way I will show you how to compile (if necessary), configure, and run each piece of the puzzle.

When you finish this chapter you'll have a deeper understanding of how collections are used in real-world applications, and adding the database and multitier application design skills to your repertoire will make you a better developer.

What You Are Going To Build

In this chapter I'm going to show you how to build a multitiered, networked, data-driven, client-server application. The application will be used to track employee training. Users can create, edit, and delete employees as well as create, edit, and delete employee training records. Employee records stored in the database will include an employee picture, so you'll need to know how to store and retrieve image data.

The overall architectural diagram for the employee training server application is given in Figure 20-1.

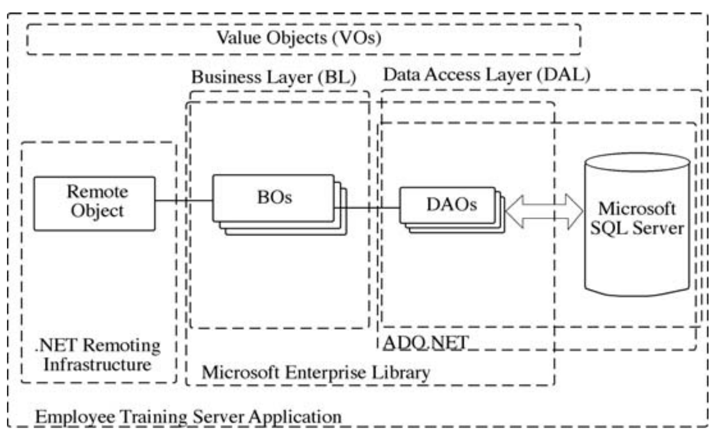


Figure 20-1: Employee Training Server Application Architecture

Referring to Figure 20-1 — the employee training server application comprises several application layers. These include the Business Layer (BL) where business objects (BOs) reside, and the Data Access Layer (DAL) where data access objects (DAOs) reside. The use of value objects (VOs) spans all application layers.

Different supporting Microsoft technologies come into play throughout the application. The Microsoft Enterprise Library can be used to support both business and data access layers, although in this chapter I am only using the Enterprise Library Data Access Application Block, which directly supports the data access layer. The .NET remoting infrastructure supports the remote object.

A *business object* is simply a class that contains the logic required to enforce the business rules of a particular application. However, try as one may to isolate *business rules* to business objects, they tend to creep into other parts of an application. For example, database design plays a key factor in what business rules can be enforced. (For example, what information about an employee is required and what information is optional?, etc.)

A business object may use the services of one or more data access objects. A *data access object* is a class whose job it is to interact directly with the database. As a rule, there is a one-to-one correspondence between data access objects and database tables. For example, an EmployeeDAO class would be responsible for interacting with the tbl_employee table in the database.

The data access layer uses the services of various classes, structures, interfaces, and enumerations provided by ADO.NET and the Microsoft Enterprise Library Data Access Application Block (DAAB). The DAAB, among other things, provides a DatabaseFactory class that is used to get a connection to the database. The DAAB also takes care of connection pooling to increase application performance when servicing multiple client connections.

The remote object supplies an interface used by remote client applications to interact with the server. The remote object uses the services provided by one or more business objects.

Referring again to Figure 20-1 — application layer dependencies radiate from right to left. The business layer depends on the data access layer, and the remote object depends on the business layer. All layers depend on the value object layer which spans all application layers.

Preliminaries

Before moving forward in this chapter you must install Microsoft SQL Server 2008 R2 and the Microsoft Enterprise Library version 5.0. You will also find it helpful to install the Microsoft SQL Server Management Studio as well, but this is not strictly required to get the application up and running. The Management Studio application provides a robust GUI interface to your SQL Server database. Both SQLServer 2008 R2 and Server Management Studio come in Express Editions free from Microsoft.

Installing SQL Server 2008 Express Edition

I use Microsoft SQL Server 2008 R2 Express Edition for the database in this chapter. Go to Microsoft's website, download the installation package and double click the installer executable file. You'll be presented with the SQL Server Installation Center window as shown in figure 20-2.

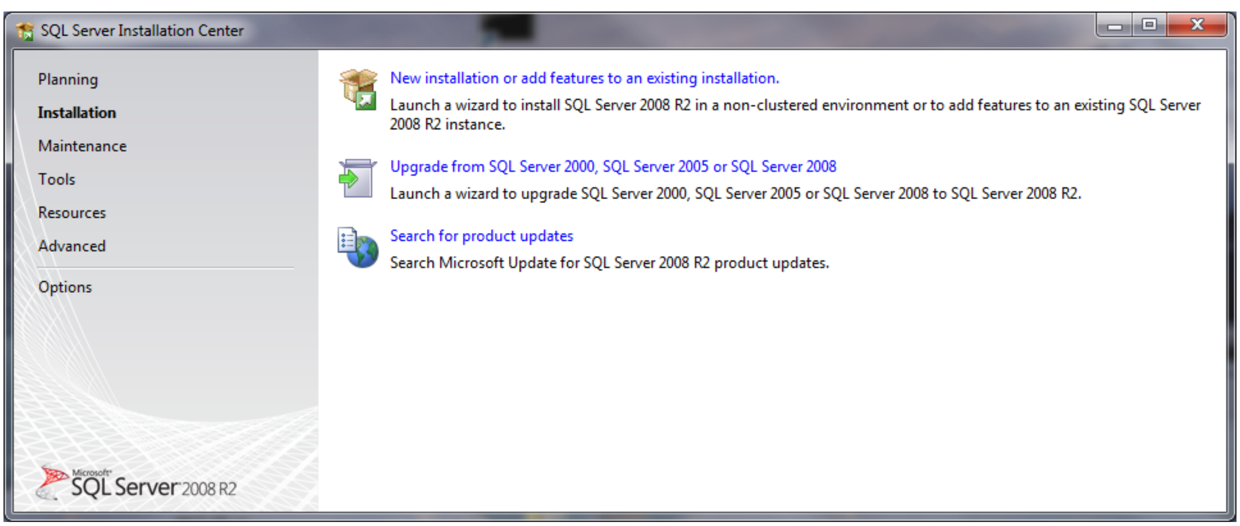


Figure 20-2: SQL Server Installation Center

Click the "New installation or add features to an existing installation." link. After several system checks you'll be presented with the SQL Server 2008 R2 Setup window as is shown in Figure 20-3.

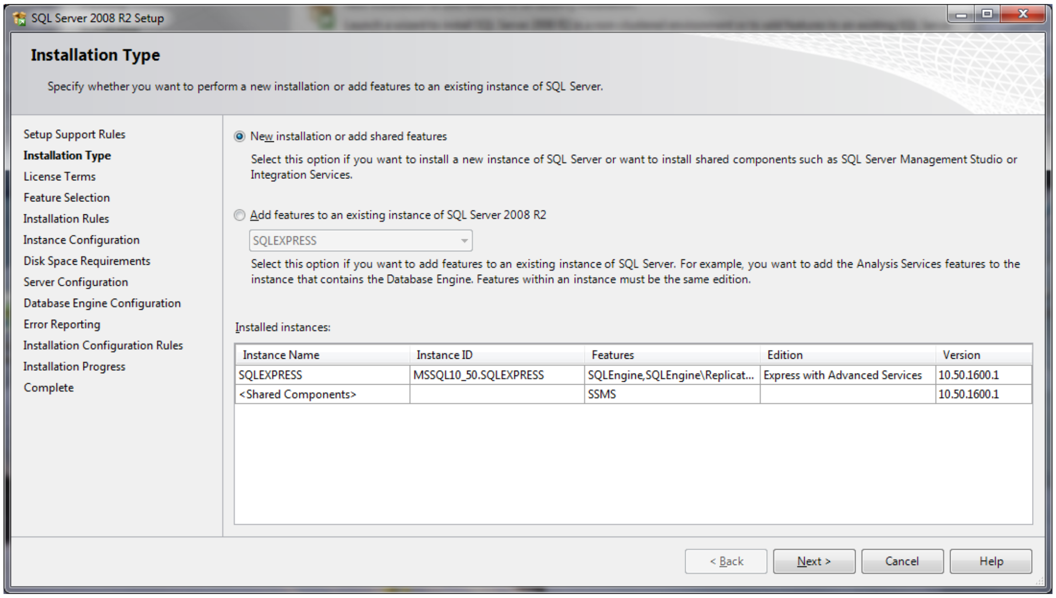


Figure 20-3: SQL Server 2008 R2 Setup Window

Click the "Next >" button to proceed with the installation.When you've finished installing SQL Server Express Edition, you can test the installation by opening a console window and entering the following command:

```
sqlcmd -S .\sqlexpress
```

This opens a connection to the default database. If all goes well you will get a line number. At the first line number "1>" enter the following SQL command:

```
select table_name from information_schema.tables
```

Press Enter. This will bring you to a second line number "2>" where you need to enter the following command:

```
go
```

Press Enter. The results you get should look similar to the output shown in Figure 20-4. To exit the SQL command prompt type the command "exit" at the line number, then press Enter.

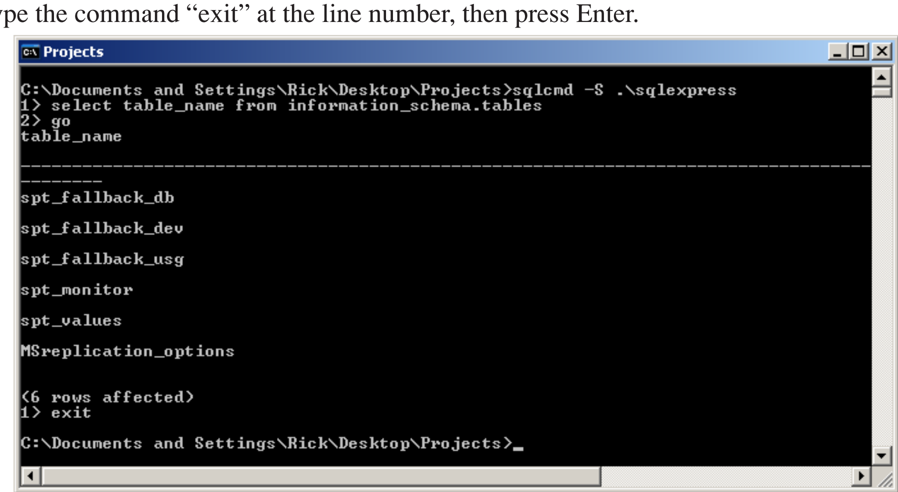


Figure 20-4: Results of Testing SQL Server Express Edition Installation

Installing Microsoft SQL Server Management Studio Express

You could do all your interaction with SQL Server Express via the SQL command utility, however, this can be cumbersome for beginners (and experienced developers too!). SQL Server Management Studio is a GUI-based application that makes it easy to manage and manipulate SQL Server databases.

You can download SQL Server Management Studio Express Edition from the same place you downloaded SQL Server Express. Follow the installation instructions and go with the default values. Installation is quick and painless. When you've finished, start SQL Server Management Studio. This will display a login dialog window similar to the one shown in Figure 20-5.

Figure 20-5: Management Studio Login Dialog

Referring to Figure 20-5 — click the Connect button to connect to the designated Server name. If you have just installed SQL Server Express there will be only one server on the list. When you click the Connect button, you'll be logged into the server and your next window will look similar to the one shown in Figure 20-6.

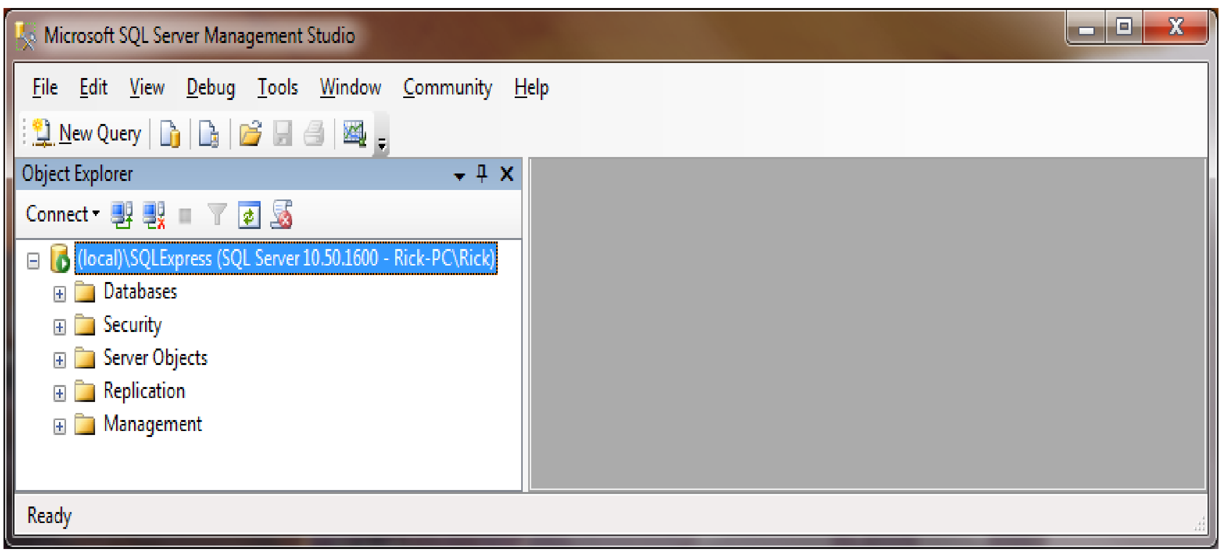


Figure 20-6: SQL Management Studio Main Window

Installing Microsoft Enterprise Library

The final thing you need to install is the Microsoft Enterprise Library version 5.0. In the interest of full disclosure, you don't need the enterprise library data application blocks to do ADO.NET programming. They just make ADO.NET programming easier to do. For the purposes of this chapter, the Enterprise Library Data Access Application Block is required.

Download the Microsoft Enterprise Library 5.0 installer from the Microsoft Patterns and Practices site. Run the enterprise library installer and click the Next button. Accept the terms of the licensing agreement and click the Next button. The next window lists the Enterprise Library 5.0 system requirements. Click the Next button. The second window you'll see will be the Custom Setup dialog window similar to the one shown in Figure 20-7.

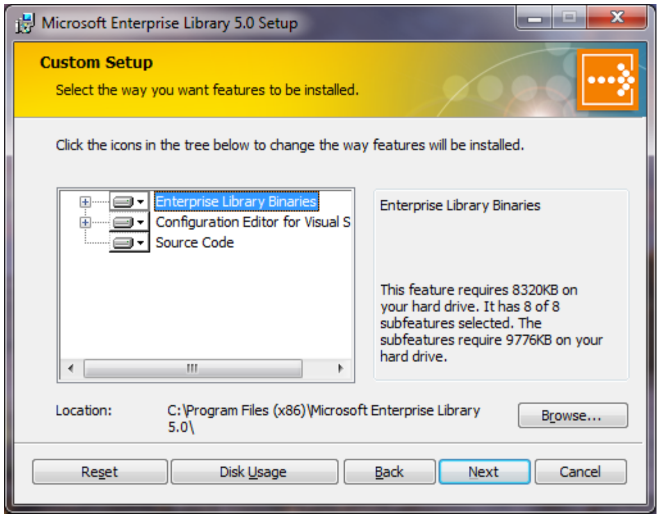


Figure 20-7: Enterprise Library Custom Setup Dialog

Referring to Figure 20-7 — accept the default installation by clicking the Next button. Installation will proceed fairly quickly. Click the Finish button on the final window to complete the setup. Verify the installation by navigating to the Microsoft Enterprise Library 5.0\Bin directory and check for the required dlls. Figure 20-8 shows a partial listing of my Microsoft Enterprise Library 5.0\Bin directory.

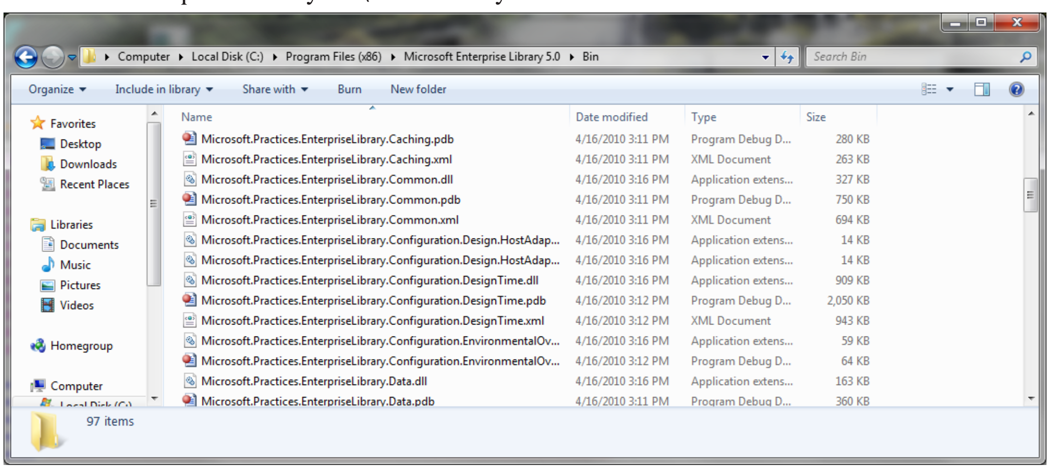


Figure 20-8: Microsoft Enterprise Library 5.0\Bin Directory Partial Listing

In this chapter you'll be using the following five enterprise library .dll files:

- Microsoft.Practices.EnterpriseLibrary.Common.dll
- Microsoft.Practices.EnterpriseLibrary.Data.dll
- Microsoft.Practices.ServiceLocation.dll
- Microsoft.Practices.Unity.dll
- Microsoft.Practices.Unity.Interception.dll

A Simple Test Application

This section presents a short and simple test application that will make sure you've got everything installed correctly. Don't proceed past this point until you get this application to run. When you're successful, you can rest assured you've got this chapter half licked!

Example 20.1 gives the code for a short application named SimpleConnection that uses a DatabaseFactory to create a Database object, and then executes a simple SQL command against that database.

20.1 SimpleConnection.cs

```
1   using System;
2   using System.Data;
3   using System.Data.Common;
4   using System.Data.Sql;
5   using System.Data.SqlClient;
6
7   using Microsoft.Practices.EnterpriseLibrary.Common;
8   using Microsoft.Practices.EnterpriseLibrary.Data;
9   using Microsoft.Practices.EnterpriseLibrary.Data.Sql;
10
11  public class SimpleConnection {
12    public static void Main(){
13      Console.WriteLine("Simple Connection!");
14      Database database = DatabaseFactory.CreateDatabase();
15      Console.WriteLine("Database created!");
16      DbCommand command = database.GetSqlStringCommand("select table_name from information_schema.tables");
17      IDataReader reader = database.ExecuteReader(command);
18      while(reader.Read()){
19        Console.WriteLine(reader.GetString(0));
20      }
21    } // end Main()
22  } // end class definition
```

Referring to Example 20.1 — note first the namespaces required. On line 14, the DatabaseFactory.CreateDatabase() method is called to create a Database object. At this point you should be wondering from where on earth does the DatabaseFactory class get the information required to create the Database object? The answer is — from a configuration file, which you'll see shortly.

On line 16, the Database object's GetSqlStringCommand() method is used to create a DbCommand object. The string used as an argument to the GetSqlStringCommand() method is a short SQL select statement, just like the one you used earlier to test the installation of SQL Server Express. The command is executed via a call to the Database object's ExecuteReader() method using the reference to the newly created Command object as an argument. It returns an IDataReader object which you use to access the query results in the body of the `while` loop. The output of this program will be a list of table names like that obtained originally in Figure 20-4.

Example 20.2 shows the contents of the simpleconnection.exe.config file.

20.2 simpleconnection.exe.config

```
<configuration>
   <configSections>
        <section name="enterpriseLibrary.ConfigurationSource"
            type="Microsoft.Practices.EnterpriseLibrary.Common.Configuration.ConfigurationSourceSection,
            Microsoft.Practices.EnterpriseLibrary.Common,
            Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
            requirePermission="true" />
        <section name="dataConfiguration"
            type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
            Microsoft.Practices.EnterpriseLibrary.Data,
            Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
            requirePermission="true" />
   </configSections>
   <enterpriseLibrary.ConfigurationSource selectedSource="System Configuration Source">
        <sources>
            <add name="System Configuration Source"
            type="Microsoft.Practices.EnterpriseLibrary.Common.Configuration.SystemConfigurationSource,
            Microsoft.Practices.EnterpriseLibrary.Common,
            Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
        </sources>
   </enterpriseLibrary.ConfigurationSource>
   <dataConfiguration defaultDatabase="Connection String" />
      <connectionStrings>
        <add name="Connection String"
        connectionString="Data source=(local)\SQLEXPRESS;Initial Catalog=master;Integrated Security=true"
        providerName="System.Data.SqlClient" />
   </connectionStrings>
</configuration>
```

Referring to Example 20.2 — the configuration file provides database connection string information. You create these configuration files with the help of the Enterprise Library Configuration tool, which you'll find in the enterprise library's installation directory. A screen shot showing the tool in action is shown in Figure 20-9. At this point it would be easier for you to either download this configuration file from the pulpfreepress.com website or create it manually by copying it from the example above.

Alright — you have the SimpleConnection.cs file and the simpleconnection.exe.config file. Before you compile the application, you'll need to copy the required Enterprise Library dll files into your project directory. Your project directory should look similar to the one shown in Figure 20.10.

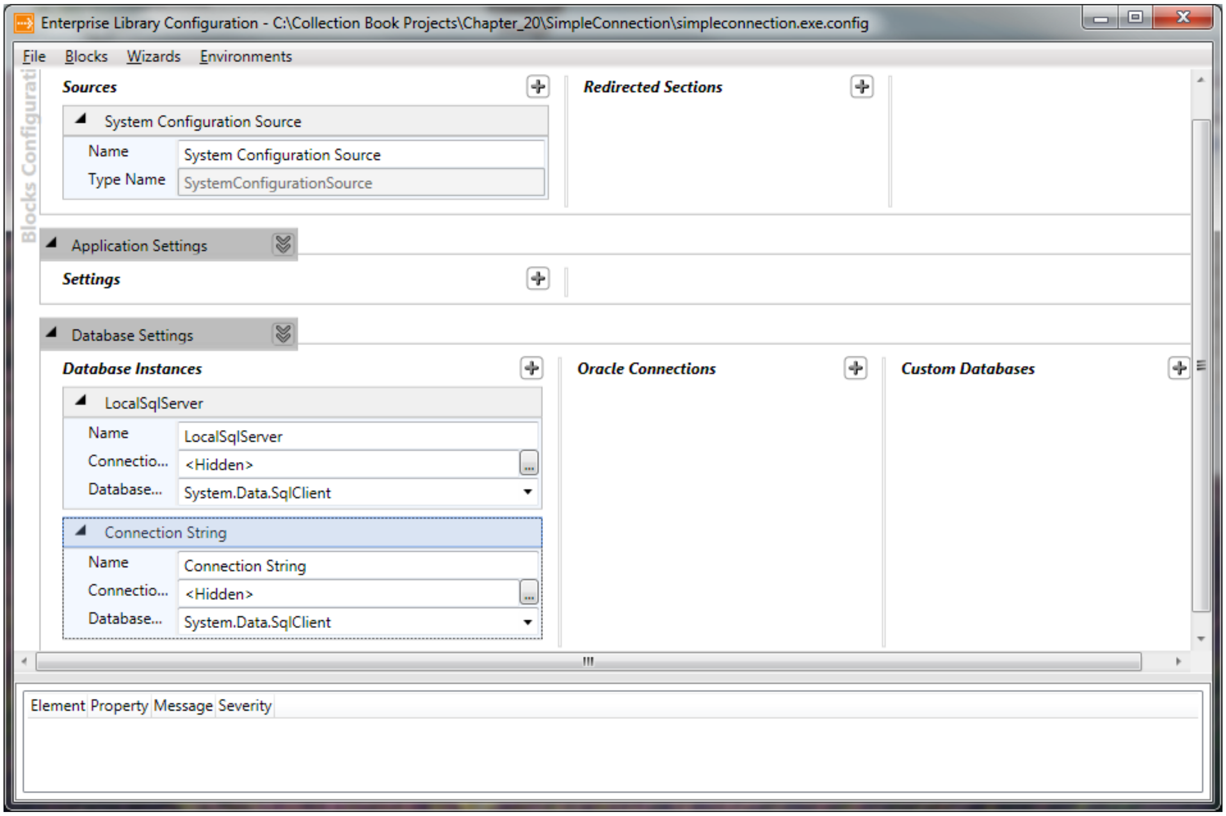


Figure 20-9: Enterprise Library Configuration File Creation Tool

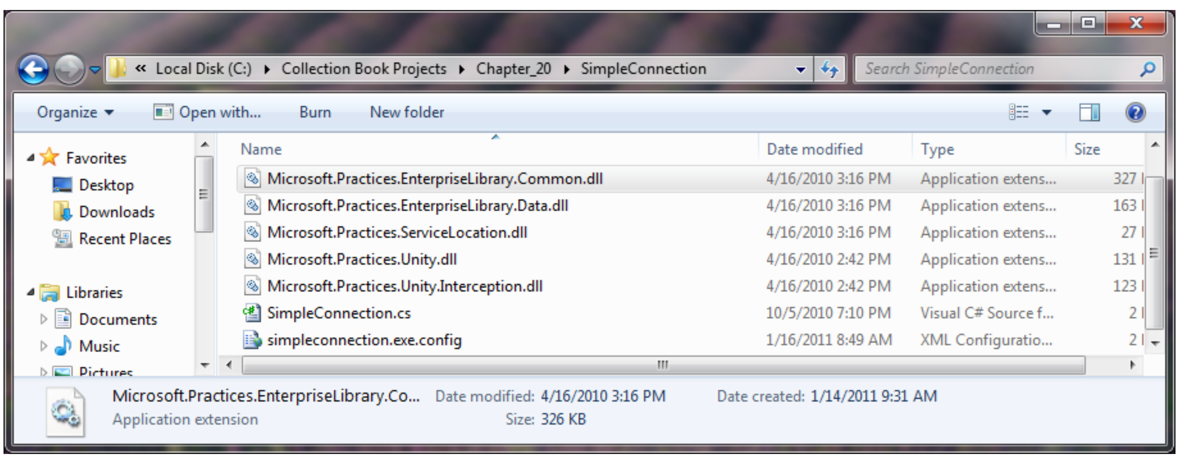


Figure 20-10: Contents of the SimpleConnection Project Directory Before Compiling

To compile this application, open a command console window, change to the project directory, and enter the following compiler command:

```
    csc /r:Microsoft.Practices.EnterpriseLibrary.Data.dll;Microsoft.Practices.EnterpriseLibrary.Common.dll /
lib:"C:\Program Files (x86)\Microsoft Enterprise Library 5.0\Bin" *.cs
```

Note that this is all on one line that has wrapped around. When you've entered this command, press Enter and cross your fingers. If all goes well it will compile. If not, you'll need to retrace your steps to ensure you've installed the database and the enterprise library files correctly. It may help to copy this rather long compiler command into a batch file named compile.bat.

Finally, run the application by typing simpleconnection at the command prompt. You should see an output similar to what is shown in Figure 20-11.

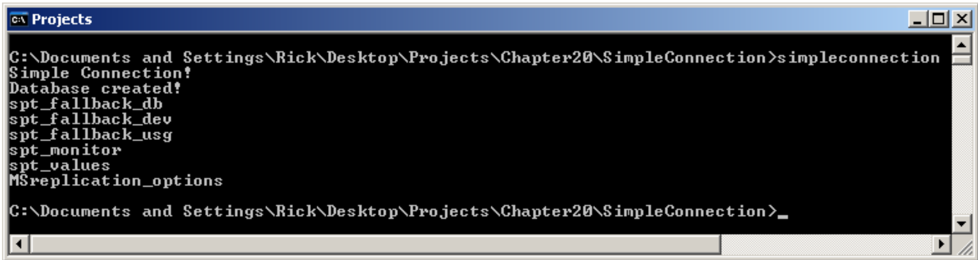


Figure 20-11: Results of Running the SimpleConnection Application

Introduction To Relational Databases And SQL

In this section I will show you how to create and manipulate data contained in a relational database using Structured Query Language (SQL). You'll also learn how to create SQL scripts to automate the execution of complex queries and other commands.

Terminology

A *database management system* (DBMS) is a software application that stores data in some form or another and provides a suite of software components that allows users to create, manipulate, and delete the data. The term *database* refers to a related collection of data. A DBMS may contain one or more databases. The term database is often used interchangeably to refer both to a DBMS and to the databases it contains. For example, a colleague is more likely to ask you "What type of database are you going to use?" rather than "What type of database management system are you going to use?"

A *relational database* stores data in relations referred to as *tables*. A *relational database management system* (RDBMS) is a software application that allows users to create and manipulate relational databases. Popular RDBMS systems that I'm personally familiar with include Oracle, MySQL, and Microsoft SQL Server, but there exist many more.

A table is composed of *rows* and *columns*. Each column has a *name* and an associated database *type*. For example, a table named tbl_employee may have a column named FirstName with a type of varchar(50). (*i.e.,* A variable-length character field with a 50 character limit.) Each row is an instance of data stored in the table. For example, the tbl_employee table might contain any number of employee entries, with each entry occupying a single row in the table.

In most cases it is desirable to be able to uniquely identify each row of data contained within a table. To do this, one or more of the table columns must be designated as the *primary key* for that table. The important characteristic of a primary key is that its value must be unique for each row.

The power of relational databases derives from their ability to dynamically create associations between different tables. One table can be related to another table by the implementation of a *foreign key*. The primary key of one table serves as the foreign key in the related table, as Figure 20-12 illustrates.

Referring to Figure 20-12 — the EmployeeID column serves as the primary key for tbl_employee. An EmployeeID column in tbl_employee_training serves as a foreign key for that table. In this manner, a relationship has been established between tbl_employee and tbl_employee_training. These tables can now be manipulated together to extract meaningful data regarding employees and the training they have taken. A table can be related to multiple tables by the inclusion of multiple foreign keys.

Primary keys and foreign keys can be used together to enforce *referential integrity*. For example, you should not be able to insert a new row into tbl_employee_training unless the EmployeeID foreign key value you are trying to insert already exists as a primary key in tbl_employee. Also, what should happen when an employee row is deleted from tbl_employee? A *cascade delete* can automatically delete any related records in tbl_employee_training. When

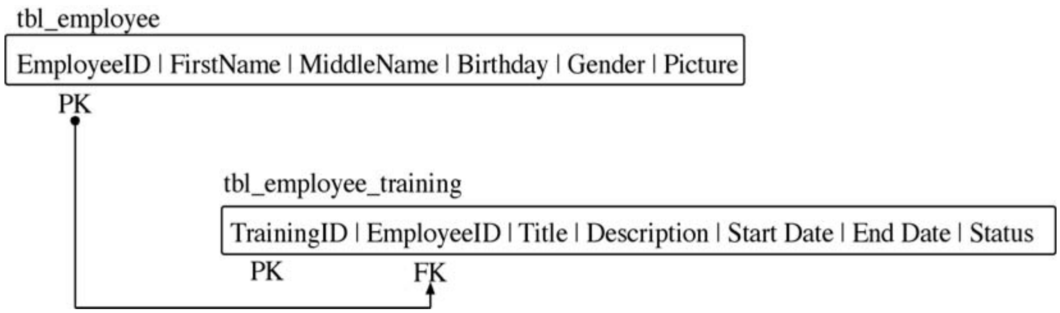


Figure 20-12: The Primary Key of One Table Can Serve as the Foreign Key in a Related Table

an employee row is deleted from tbl_employee, any rows in tbl_employee_training with a matching foreign key will also be deleted.

Structured Query Language (SQL)

SQL is used to create, manipulate, and delete relational database objects and the data they contain. Although SQL is a standardized database language, each RDBMS vendor is free to add extensions to the language, which essentially renders the language non-portable between different database products. What this means to you is that while the examples I present in this section will work with Microsoft SQL Server, they may not work with Oracle, MySQL, or whatever relational database system you're familiar with. This holds true especially for SQL's Data Definition Language commands, which we will cover shortly.

SQL comprises three sub-languages, which group commands according to functionality: Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL). In this section I will focus on the use of DDL and DML.

Data Definition Language (DDL)

The DDL includes the `create`, `use`, `alter`, and `drop` commands. Let's use a few of these commands to set up the employee training database that will be used to store data for the employee training application. Before we begin, open SQL Server Management Studio and take a look at the default databases SQL Server provides upon installation.

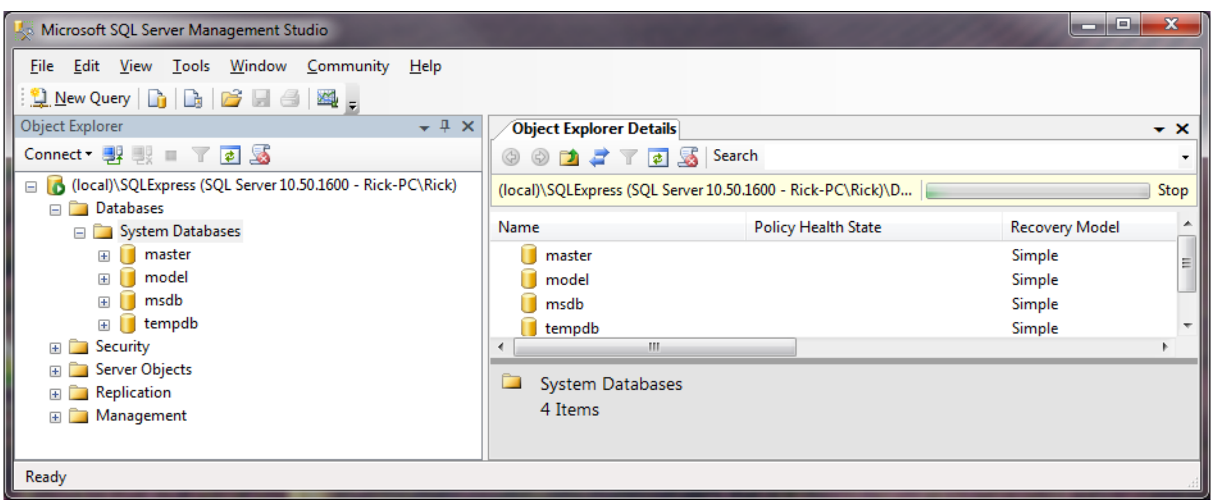


Figure 20-13: SQL Server's Default Databases

Referring to Figure 20-13 — there are four databases installed by default. These include *master*, *model*, *msdb*, and *tempdb*. Of these four, the *master* database is most important. It contains data necessary to startup and run SQL

Server. Ordinarily, you will not directly interface with or manipulate the master database, but you will need to use it every once in a while with the `use` command as you will see shortly.

Creating The EmployeeTraining Database

Let's now create the EmployeeTraining database with the help of the `create` command. You could create the database using Management Studio, but I want to show you how to do it using the SQL command utility and then with the help of an SQL script file.

First, open a command window and start the SQL command utility with the following command:

```
sqlcmd -S .\sqlexpress
```

On the first numbered line enter the following command:

```
create database EmployeeTraining
```

Press Enter, and on the second numbered line enter the following command:

```
go
```

Press Enter. Your command window should look similar to Figure 20-14.

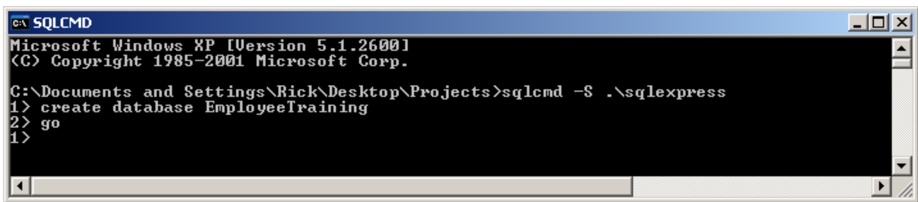


Figure 20-14: Creating EmployeeTraining Database with SQL Command Utility

Check that the database exists by opening Management Studio and taking a look. You should see something similar to Figure 20-15.

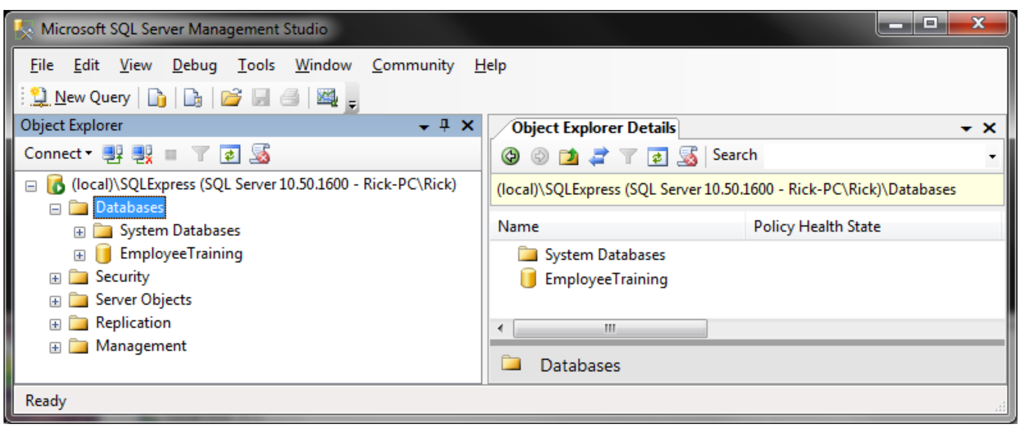


Figure 20-15: Checking on the Existence of the EmployeeTraining Database

Creating A Database With A Script

Alright, now that you've done this via the SQL command utility line-by-line, I want to show you how to drop the database and create it with a script. Open your favorite text editor and create a file named "create_database.sql" and enter the code shown in Example 20.3

20.3 create_database.sql

```
use master
drop database EmployeeTraining
go

create database EmployeeTraining
go
```


Referring to Example 20.3 — one line 1, the `use` command switches to the master database context. The `drop` command on line 2 drops the EmployeeTraining database. The `go` command on line 3 executes the previous two lines. On line 5, the `create` command creates the EmployeeTraining database. The `go` command is used again on line 6 to execute line 5.

Save the create_database.sql file in a folder named "scripts". In fact, now would be a good time to create a project folder for the employee training application. I recommend two folders: one named "client", the other named "server". Create the scripts folder in the server folder.

To execute the create_database.sql script, change to the scripts folder and enter the following command:

```
sqlcmd -S .\sqlexpress -i create_database.sql
```

Press Enter. If all goes well, you'll see an output similar to that shown in Figure 20-16.

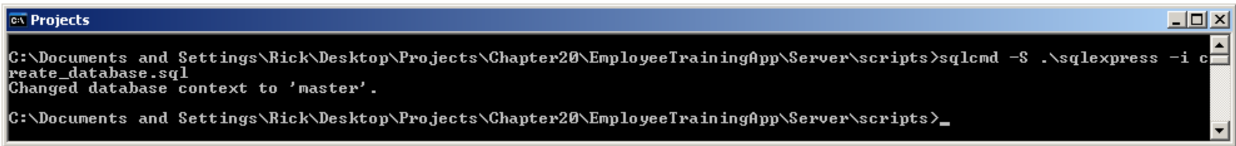


Figure 20-16: Results of Executing the create_database.sql Script

As you can see from looking at Figure 20-16, there's not much output, only one line indicating the database context changed to master. Open Management Studio and verify once again that the EmployeeTraining database exists. It's now time to create the tables we'll use to store the employee training application data.

Creating Tables

The `create` command is used to create the tables we'll need to store data for employees and their training. Now, while you could create the tables via the SQL command utility line-by-line, that method is error-prone and hard to edit. It's much easier to create a script to do the work for you. Example 20.4 gives the first version of a database script named "create_tables.sql" that contains the SQL code required to create a table named "tbl_employee".

20.4 create_tables.sql (1st version)

```
1   use EmployeeTraining
2
3   drop table tbl_employee
4   go
5
6   create table tbl_employee (
7      EmployeeID uniqueidentifier not null primary key,
8      FirstName varchar(50) not null,
9      MiddleName varchar(50) not null,
10     LastName varchar(50) not null,
11     Birthday datetime not null,
12     Gender varchar(1) not null,
13     Picture varbinary(MAX) null
14  )
15  go
```

Referring to Example 20.4 — it's imperative that this script executes in the EmployeeTraining database, and that's the purpose of the `use` command on line 1. Line 3 drops the tbl_employee table, if it exists. It certainly will *not* exist the first time you execute the script, so you'll see an error message stating that fact. You can safely ignore that message. The create command starting on line 6 creates the tbl_employee table. The tbl_employee table contains seven columns named *EmployeeID*, *FirstName*, *MiddleName*, *LastName*, *Birthday*, *Gender*, and *Picture*. Each column has a corresponding database type. Most are of the variable length character type varchar(*n*) where *n* specifies the maximum number of characters the column can contain. The EmployeeID column is of type *uniqueidentifier* which has been designated as the table's primary key column. The Birthday column is of type datetime, and the Picture column is a variable length binary column set to varbinary(MAX). All columns except Picture must contain data when a row is created. This is specified with the `not null` constraint. (A *constraint* is a rule placed on a column or table meant to enforce data integrity.)

In this example application the tbl_employee table is fairly simple and straightforward. I may, in the not to distant future, regret the decision to put the employee picture in the tbl_employee table, but for now that's where I'm putting it!

To run this script save it in the scripts folder, open a command window, change to the scripts folder, and enter the following command-line command:


```
sqlcmd -S .\sqlexpress -i create_tables.sql
```

The results obtained from executing this script on my machine are shown in Figure 20-17.

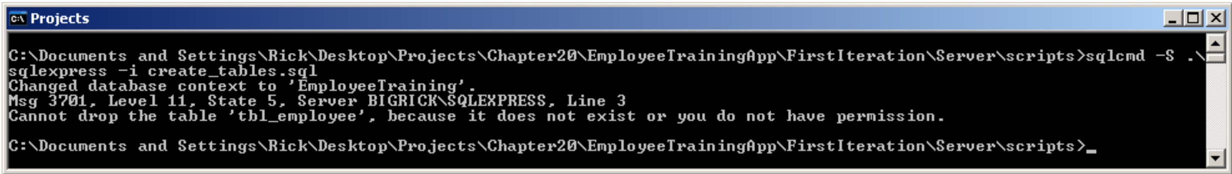


Figure 20-17: Results of Executing create_tables.sql Database Script

SQL Server Database Types

Table 20-1 lists the MS SQL Server database types and their associated value ranges and usage.

Type Category	Data Type	Value Range	Usage
Exact Numeric	bigint	-2^{63} to $2^{63}-1$	Use to store large integral values.
Exact Numeric	int	-2^{31} to $2^{31}-1$	Use to store medium-sized integral values.
Exact Numeric	smallint	-2^{15} to $2^{15}-1$	Use to store small integral values.
Exact Numeric	tinyint	0 to 255	Use to store really small integral values.
Exact Numeric	bit	0, 1, or null	Stores 1 or 0
Exact Numeric	decimal	$-10^{38}+1$ to $10^{38}-1$	decimal(p, s) where p is precision and s is scale. P is the maximum total number of decimal digits that can be stored both to the left and right of the decimal point. The range of p is 1 - 38 with 18 as the default. Scale is the maximum number of decimal digits that can be stored to the right of the decimal point. The range of s varies from 0 - p. (Example decimal(24, 6) would specify 24 total digits with 6 to the right of the decimal point.)
Exact Numeric	numeric	$-10^{38}+1$ to $10^{38}-1$	numeric is equivalent to decimal
Exact Numeric	money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	Large monetary or currency values. Use to hold the value of US national debt.
Exact Numeric	smallmoney	-214,748.3648 to 214,748.3647	Small monetary or currency values.
Approximate Numerics	float	-1.79^{308} to -2.23^{-308}, 0, and 2.23^{-308} to 1.79^{308}	float(n) where n is the number of bits used to store the mantissa. n must be a value between 1 - 53. Default value of n is 53.
Approximate Numerics	real	-3.40^{38} to -1.18^{-38}, 0 and 1.18^{-38} to 3.4^{38}	Equivalent to float

Table 20-1: SQL Server Data Types

Type Category	Data Type	Value Range	Usage
Date and Time	datetime	1 January 1753 through 31 December 9999	Holds a large date and time range.
Date and Time	smalldatetime	1 January 1900 through 6 June 2079	Holds a smaller date and time range.
Character Strings	char	1 - 8000 fixed length bytes	Holds fixed length character values.
Character Strings	varchar	1 - 8000 variable length bytes or varchar(MAX) holds 2^{31}-1 bytes	Holds variable length character strings varchar(n) where n specifies max length.
Character Strings	text	DO NOT USE	Will be removed from future versions of SQL Server
Unicode Character Strings	nchar	1 - 4000 fixed length unicode characters	Holds fixed length unicode character strings.
Unicode Character Strings	nvarchar	1 - 4000 variable length unicode characters or nvarchar(MAX) holds 2^{31}-1 bytes	Holds variable length unicode character strings. nvarchar(n) where n specifies max length.
Unicode Character Strings	ntext	DO NOT USE	Will be removed from future versions of SQL Server
Binary Strings	binary	1 - 8000 fixed length binary data	Holds fixed length binary data.
Binary Strings	varbinary	1 - 8000 variable length binary data or varbinary(MAX) holds 2^{31}-1 bytes	Holds variable length binary data. varbinary(n) where n specifies max length.
Binary Strings	image	DO NOT USE	Will be removed from future versions of SQL Server
Other	cursor	cursor reference	Holds variables or stored procedure output parameters that contain a reference to a cursor.
Other	sql_variant	int, binary, and char	Stores values of various data types.
Other	table	result set	Stores a result set for later processing.
Other	timestamp	Automatically generated unique binary number	Used to version-stamp table rows. Does not preserve a date or a time.
Other	uniqueidentifier	A 16-byte Globally Unique Identifier (GUID)	Used to hold GUID strings.
Other	xml	2 gigabytes	Holds XML data.

Table 20-1: SQL Server Data Types

Referring to Table 20-1 — note that three database types have been deprecated and will, at some point in the future, be dropped from SQL Server. These have been highlighted with light grey shading. Now, while this may or may not happen for a long, long time, it's still a good idea to shy away from using the deprecated types when writing new code.

Data Manipulation Language (DML)

Now that you've created the EmployeeTraining database and added to it the tbl_employee table, it's time to learn how to use SQL's Data Manipulation Language to add, manipulate, and delete tbl_employee data. There are four

DML commands: `insert`, `select`, `update`, and `delete`. First things first! Let's create a script to insert some test data into the tbl_employee table. I'll then show you how to manipulate that data with the other three commands.

Using The Insert Command

Example 20.5 gives the code for a database script named "create_test_data.sql" that inserts one row of test data into the tbl_employee table.

20.5 create_test_data.sql

```
1   use EmployeeTraining
2   go
3
4   insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
5   values (newid(), 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
6   go
```

Referring to Example 20.5 — line 1 is necessary to ensure we are using the correct database, which in this case is EmployeeTraining. Line 4 contains the first part of the insert statement. With the `insert` statement you specify into which table you want to insert data, and list each column that will receive data in the parentheses. The order of the columns you specify is important because the order of the actual values you insert, shown here on line 5, must match the order in which you listed your columns. In this example, I am inserting data into the employeeid, firstname, middlename, lastname, birthday, and gender columns only. Remember, these columns MUST contain data because of their NOT NULL constraint. It's ok not to insert data into the picture column because that column is allowed to contain a null value. (**Note:** If you want to insert more than one row of test data simply add another insert statement to the script below line 6.)

To execute this script, open a command window, change to the scripts folder, and enter the following command:

```
sqlcmd -S .\sqlexpress -i create_test_data.sql
```

Then press Enter. You should see a result similar to that shown in Figure 20-18.

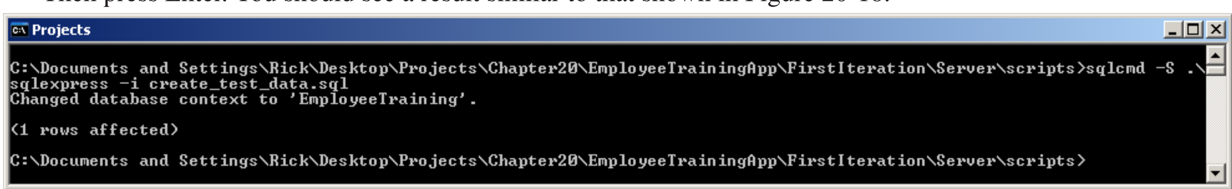


Figure 20-18: Results of Running create_test_data.sql Database Script

Using The Select Command

The `select` command is used to write database queries (*i.e.,* select statements) that return data. A `select` statement contains several clauses, most of which are optional. The following code fragment shows a simple `select` statement that gets all the data contained in all the columns of the tbl_employee table:

```
select * from tbl_employee
```

To execute this `select` statement, open the SQL command utility by typing the following command-line command:

```
sqlcmd -S .\sqlexpress
```

At the first numbered line enter the following command:

```
use employeetraining
```

Press Enter, then enter `go` and press Enter again. On the first numbered line enter the `select` command given above and press Enter. On the next numbered line enter the `go` command and press Enter. Your results should look similar to Figure 20-19.

Referring to Figure 20-19 — the output is a little bunched up but you can pick out the column headings and their associated data.

You can limit the number of columns a select statement returns by specifying exactly which columns you want when you enter the select statement, as the following code fragment shows:

```
select firstname, middlename, lastname from tbl_employee
```

Try executing this statement in the SQL command utility. Your results should look similar to those shown in Figure 20-20.

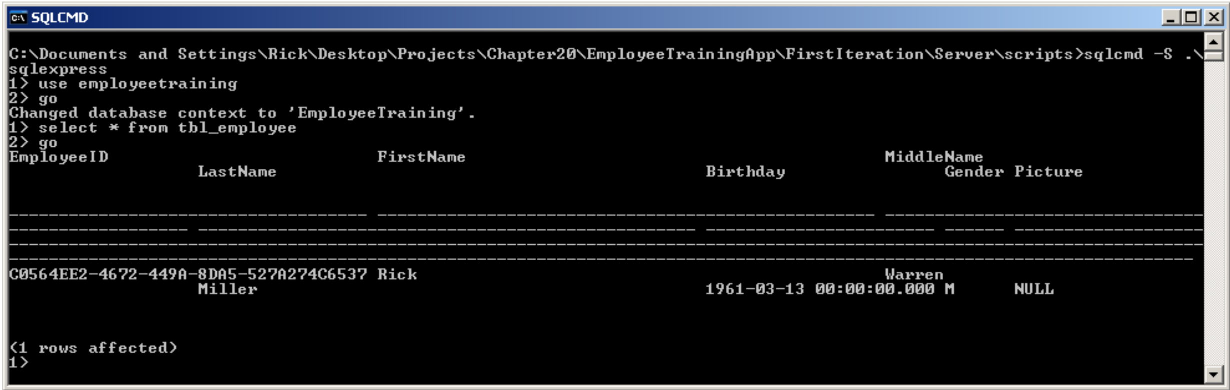


Figure 20-19: Results of Executing a Simple Select Statement

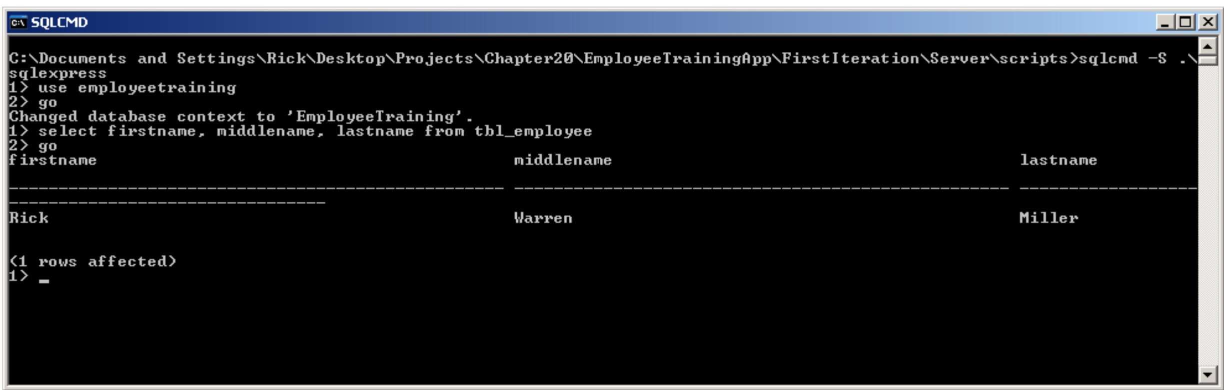


Figure 20-20: Selecting Specific Rows with select Statement

Up to this point I've only been using one required `select` statement `from` clause to specify the table from which to get the data. The following `select` statement adds an optional `where` clause to limit the data returned:

```
select firstname, lastname from tbl_employee where lastname='Bishop'
```

As you may have guessed, if you entered this query in the employeetraining database, you'd get no results because nobody by the last name of Bishop has been entered into the tbl_employee table. Let's modify the create_test_data.sql script to add some more test data. Example 20.6 gives the modified script.

20.6 create_test_data.sql (Mod 1)

```
use EmployeeTraining
go

insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Steve', 'Jacob', 'Bishop', '2/10/1942', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Coralie', 'Sarah', 'Powell', '10/10/1974', 'F')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Kyle', 'Victor', 'Miller', '8/25/1986', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Patrick', 'Tony', 'Condemi', '4/17/1961', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values (newid(), 'Dana', 'Lee', 'Condemi', '11/1/1965', 'F')
go
```

Referring to Example 20.6 — notice how the `go` command must be issued after each `insert` statement. Run this script to insert the extra data. (**Note:** You may want to run the create_tables.sql script to start clean! Don't you just love database scripts!) Your results should look similar to Figure 20-21.

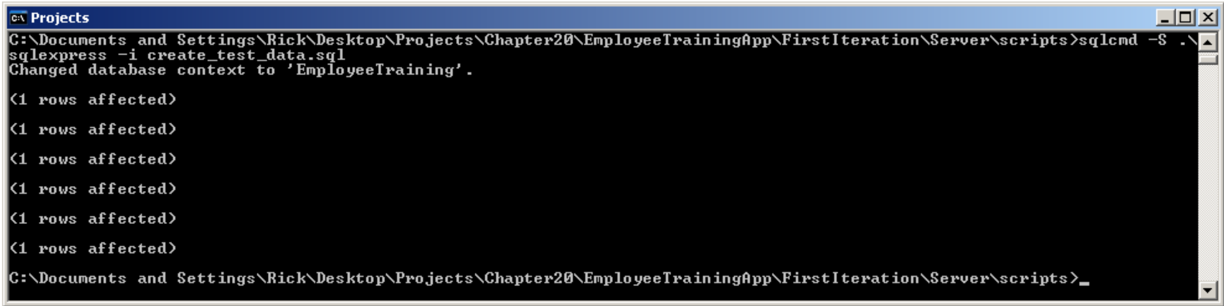


Figure 20-21: Inserting More Test Data with the create_test_data.sql Database Script

Now, start the SQL command utility, change to the employeetraining database, and enter the following `select` statement:

```
select firstname, lastname
from tbl_employee
where lastname='Miller'
go
```

This is how you will normally see a `select` statement used, with each clause appearing on separate lines. Note that the `go` command is not part of SQL, but rather how the SQL statement is executed in SQL Server. The results you get from executing this query should look similar to those shown in Figure 20-22.

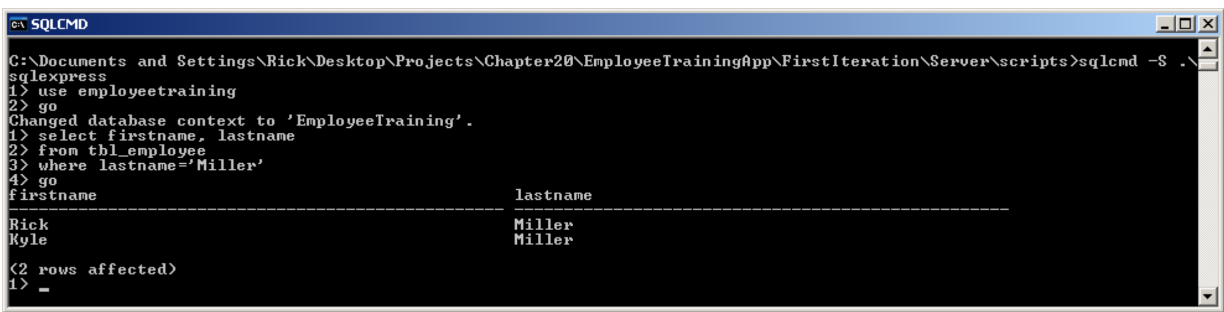


Figure 20-22: Results of Limiting Data Returned from select Statement with *where* Clause

Referring to Figure 20-22 — the query returned two employees with the last name Miller. If you did not run the create_tables.sql script before running the modified create_test_data.sql script, you would see three employees in the results because there would be two Rick Millers in the database.

Try this query:

```
select firstname, lastname
from tbl_employee
where gender='F' or firstname='Kyle'
go
```

This should return three rows as is shown in Figure 20-23.

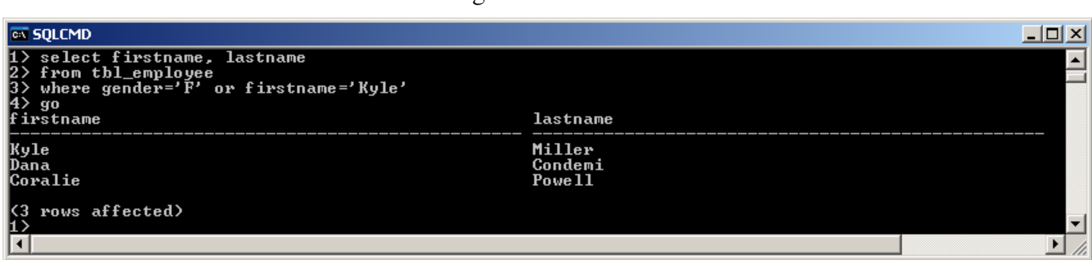


Figure 20-23: Results of Executing the Previous Query

Using The Update Command

Data within a table can be changed with the SQL `update` command. For example, if you wanted to change the employee Coralie Powell's last name to Miller, you would use the following `update` statement:

```
update tbl_employee
set lastname = 'Miller'
where firstname = 'Coralie'
go
```

The `update` statement begins by specifying the name of the table to which the update applies. The `set` clause on the second line specifies one or more columns within that table and their new values. The `where` clause is used to specify to which row in the table the update applies. In this case the employee whose first name is "Coralie" will have her name changed from "Powell" to "Miller". (**Note:** If you had more than one employee with the first name "Coralie", this statement would change all their last names to Miller. To isolate the correct Coralie you'd have to use her EmployeeID in the `where` clause.) Figure 20-24 illustrates the use of the previous `update` statement.

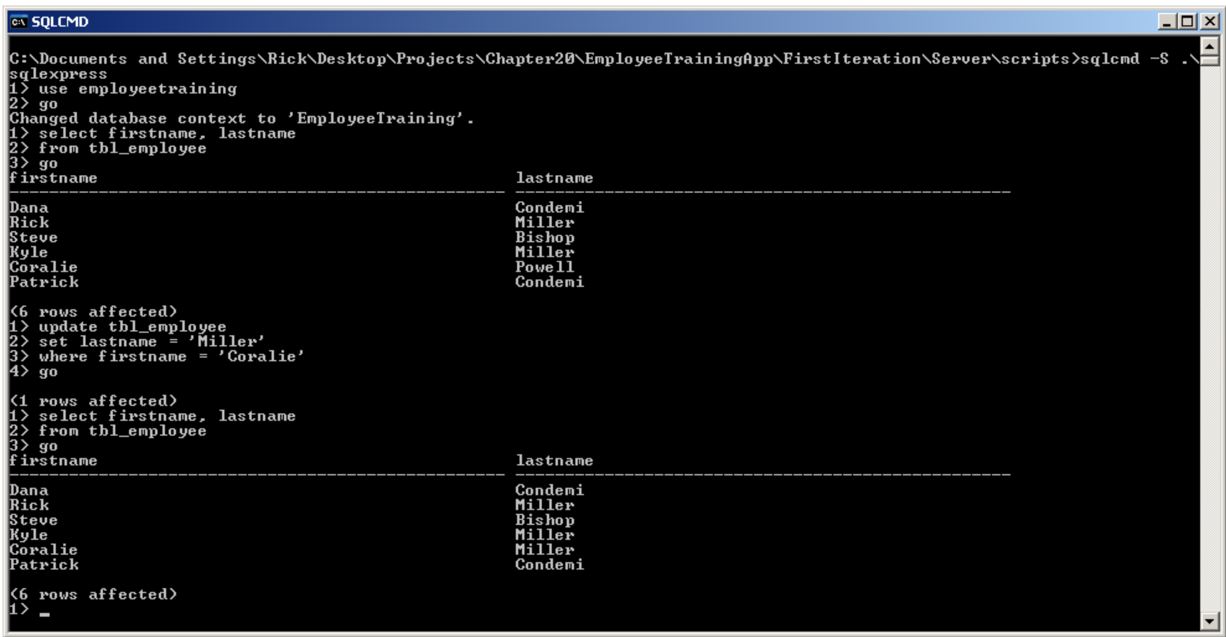


Figure 20-24: Changing Coralie Powell's Last Name to Miller with the Update Statement

Using The Delete Command

The `delete` command is used to delete one or more rows from a table. The following `delete` statement removes from the tbl_employee table all employees whose last names equal "Miller":

```
delete from tbl_employee
where lastname = 'Miller'
go
```

The results of executing this statement are shown in Figure 20-25.

Quick Review

Relational databases hold data in *tables*. Table *columns* are specified to be of a particular *data type*. Table data is contained in *rows*. Structured Query Language (SQL) is used to *create*, *manipulate*, and *delete* relational database objects and data. SQL contains three sub-languages: Data Definition Language (DDL) which is used to create databases, tables, views, and other database objects; Data Manipulation Language (DML) which is used to create, manipulate, and delete the data contained within a database; and Data Control Language (DCL) which is used to grant or revoke user rights and privileges on database objects.

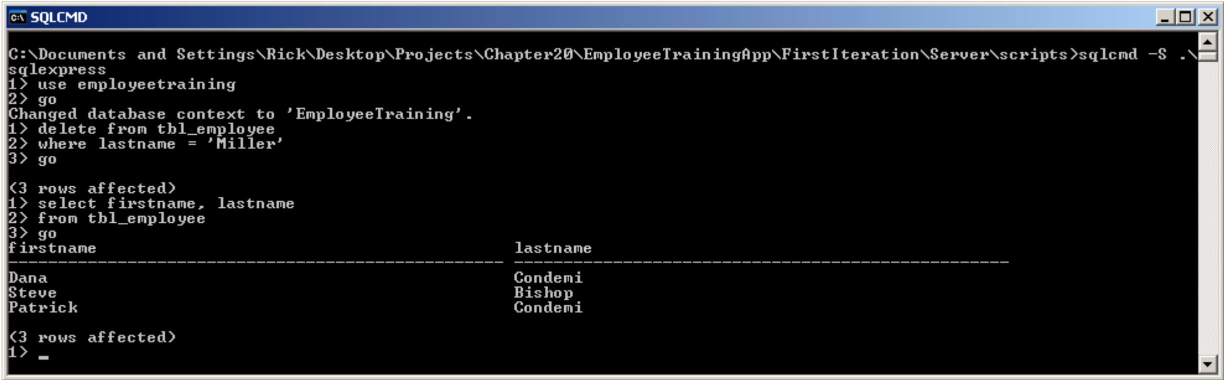


Figure 20-25: Deleting all Employees whose Last Names = "Miller"

Different database makers are free to extend SQL to suit their needs so there's no guarantee of SQL portability between different databases.

One or more table columns can be designated as a *primary key* whose value is unique for each row inserted into that table. Related tables can be created by including the primary key of one table as a *foreign key* in the related table.

Complex SQL Queries

In this section I want to show you how to use SQL to manipulate data in multiple tables. Along the way you will learn how to use a *foreign key* to create a related table, and how to use the `from` clause to *join* related tables together in a `select` statement.

Creating A Related Table With A Foreign Key

It's time now to add another table to the employeetraining database. Where do you think would be a good place to put this table's `create` statement code? If you guessed the create_tables.sql script you're right!

20.7 create_tables.sql (Mod 1)

```
use EmployeeTraining

alter table tbl_employee_training drop constraint fk_employee
go

drop table tbl_employee
go

create table tbl_employee (
   EmployeeID uniqueidentifier not null primary key,
   FirstName varchar(50) not null,
   MiddleName varchar(50) not null,
   LastName varchar(50) not null,
   Birthday datetime not null,
   Gender varchar(1) not null,
   Picture varbinary(max) null
)
go

drop table tbl_employee_training
go

create table tbl_employee_training (
   TrainingID int not null identity(1,1) primary key,
   EmployeeID uniqueidentifier not null,
   Title varchar(200) not null,
   Description varchar(500) not null,
   StartDate datetime null,
   EndDate datetime null,
   Status varchar(25)
)
go

```



```
34  alter table tbl_employee_training
35  add constraint fk_employee
36  foreign key (EmployeeID)
37  references tbl_employee (EmployeeID) on delete cascade
38  go
```

Referring to Example 20.7 — the `create` statement for the tbl_employee_training table begins on line 23. It's preceded by the `drop` statement on line 20. The table's primary key is named TrainingID. The primary key value, in this case an integer, will be automatically generated when a record is inserted into the table and incremented by 1. This behavior is obtained with the identity(1,1) entry specification. The first value is the identity seed, the second is the increment value.

Note that the tbl_employee_training table has a column named EmployeeID, which is the same type as the EmployeeID column in the tbl_employee table. However, this alone does not establish the foreign key relationship between that column and the one in the tbl_employee table. The *foreign key constraint* is created with the `alter` statement beginning on line 34. Note that the name of the foreign key constraint is fk_employee. (You could name it anything you like.) Having the foreign key constraint named in this manner allows you to drop the constraint before you drop the tbl_employee table. If you don't drop the fk_employee constraint before trying to drop the tbl_employee table you'll get an error. That's why it's necessary to put the `alter` statement on line 3.

To run this script, change to the scripts directory and enter the following command at the command-line:

```
sqlcmd -S .\sqlexpress -i create_tables.sql
```

The first time you run this script you'll get several errors saying the fk_employee constraint and tbl_employee_training table do not exist. When you run it a second time you will not receive those errors. After you run the script, verify the existence of the tbl_employee_training table by opening SQL Server Management Studio as is shown in Figure 20-26.

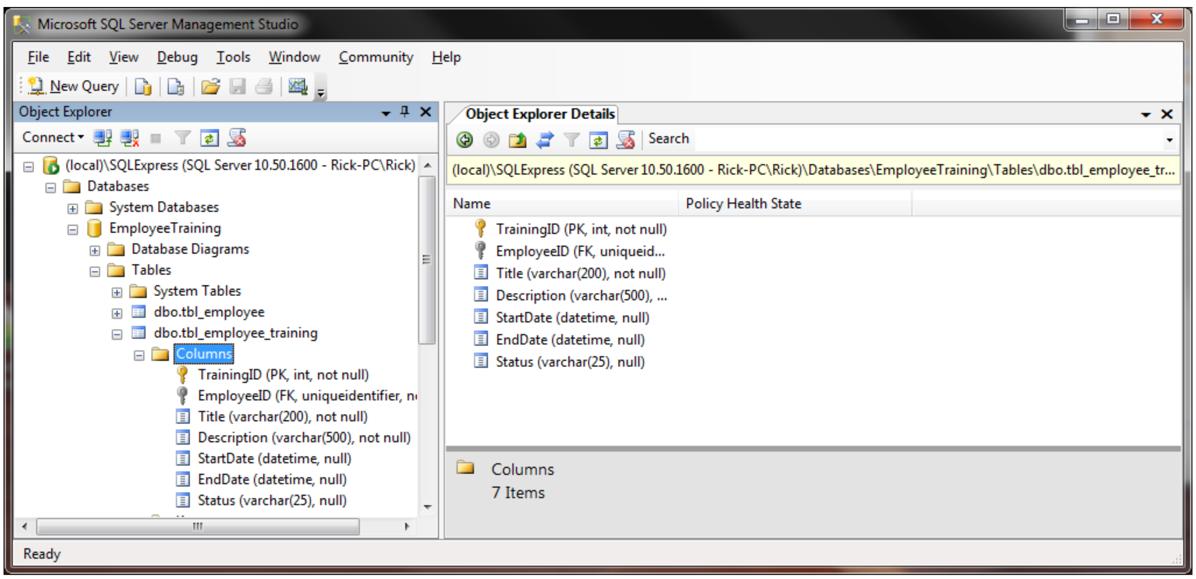


Figure 20-26: Verifying the Creation of the tbl_employee_training Table

Inserting Test Data Into The tbl_employee_training Table

You'll want to insert some test data into the tbl_employee_training table and to do this you'll need to make several modifications to the create_test_data.sql script. But first, run the script as-is to insert test data into the tbl_employee table. You need to do this so that you can get a valid GUID for each employee. To do this, run the script, then enter the following command in the SQL command utility: (Don't forget to change to the employeetraining database first!)

```
select employeeid from tbl_employee
go
```

Figure 20-27 shows this statement being executed in the SQL command utility.

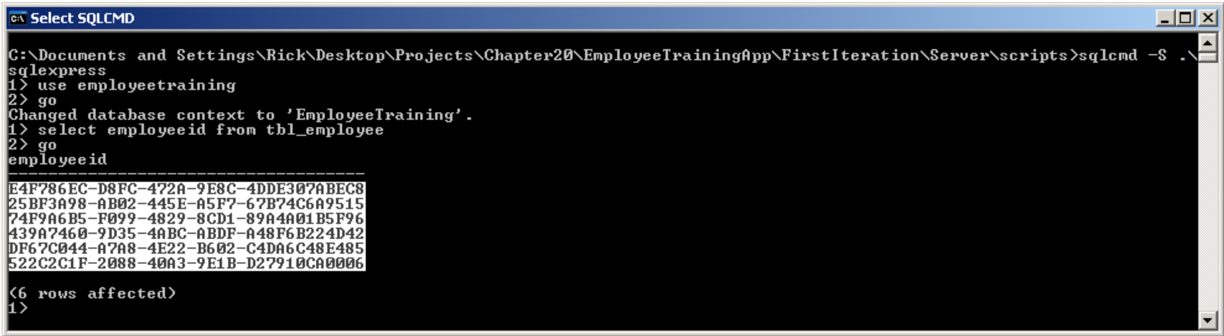


Figure 20-27: Selecting EmployeeIDs from tbl_employee

Referring to Figure 20-27 — select the listed EmployeeIDs, copy them, and paste them into your text editor. You'll need them to create the modified version of the create_test_data.sql script as is shown in Example 20.8.

20.8 create_test_data.sql (Mod 2)

```
use EmployeeTraining
go

insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Rick', 'Warren', 'Miller', '3/13/1961', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('25BF3A98-AB02-445E-A5F7-67B74C6A9515', 'Steve', 'Jacob', 'Bishop', '2/10/1942', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('74F9A6B5-F099-4829-8CD1-89A4A01B5F96', 'Coralie', 'Sarah', 'Powell', '10/10/1974', 'F')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Kyle', 'Victor', 'Miller', '8/25/1986', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('DF67C044-A7A8-4E22-B602-C4DA6C48E485', 'Patrick', 'Tony', 'Condemi', '4/17/1961', 'M')
go
insert into tbl_employee (employeeid, firstname, middlename, lastname, birthday, gender)
values ('522C2C1F-2088-40A3-9E1B-D27910CA0006', 'Dana', 'Lee', 'Condemi', '11/1/1965', 'F')
go


insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Advanced Microsoft Word', 'Description text here...',
    '11/2/2007', '11/5/2007', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8', 'Project Management Professional',
    'Description text here...', '6/12/2006', '6/15/2006', 'Passed')
go

insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('25BF3A98-AB02-445E-A5F7-67B74C6A9515', 'Project Management Professional',
    'Description text here...', '6/12/2006', '06/15/2006', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('74F9A6B5-F099-4829-8CD1-89A4A01B5F96', 'C# Programming', 'Description text here...',
    '1/15/2007', '5/8/2007', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Managing Difficult Employees',
    'Description text here...', '1/2/2007', '1/4/2007', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('439A7460-9D35-4ABC-ABDF-A48F6B224D42', 'Project Management Professional',
    'Description text here...', '6/12/2006', '6/15/2006', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('DF67C044-A7A8-4E22-B602-C4DA6C48E485', 'Squeezing Profit Margins', 'Description text here...',
    '7/5/2004', '7/10/2004', 'Passed')
go
insert into tbl_employee_training (EmployeeID, Title, Description, StartDate, EndDate, Status)
values ('522C2C1F-2088-40A3-9E1B-D27910CA0006', 'Project Financial Management',
    'Description text here...', '8/2/2007', '8/5/2007', 'Passed')
go
```


Referring to Example 20.8 — the modified create_test_data.sql statement does away with the newid() function and instead inserts employees into the tbl_employee table with hard-wired employee ids. These employee ids are then used to insert one or more training records into the tbl_employee_training table for each employee. Note that because of the foreign key constraint between the tbl_employee_training and tbl_employee tables, the `insert` statement checks to ensure the EmployeeID being inserted into tbl_employee_training is valid, meaning that the EmployeeID does in fact exist as a primary key in the tbl_employee table. If it were invalid the insert would fail.

To run this script be sure to first run the new create_tables.sql script to get rid of any data that may be in the tables. (Both tables should be empty at this point but if, in the future, you want to reset the test data, you'll get errors if you try to run this script without first deleting the data in the tbl_employee table since the EmployeeID values are hard-wired.)

Now that we have a mix of employee and training test data loaded into the database, I can show you how to use the `select` statement to create complex queries that span multiple tables.

Selecting Data From Multiple Tables

The `select` statement can be used to perform complex database queries involving multiple tables. In this section, I show you how to use the `select` statement to *join* the tbl_employee and tbl_employee_training tables together to answer complex employee training queries.

Join Operations

Related database tables can be *joined* together to answer complex database queries. There are several different types of *join* operations but the most common one is an *inner join*, which is the default SQL Server join operation.

A join operation results in a new temporary table that contains the results of the join. A join can involve any number of related tables, or non-related tables in the case of outer joins.

Let's start by listing all the training each employee has taken and sort the results by last name. This query is shown in the following `select` statement:

```
select firstname, lastname, title
from tbl_employee, tbl_employee_training
where tbl_employee.EmployeeID = tbl_employee_training.EmployeeID
order by lastname
go
```

In this example, the `from` clause implicitly joins the tbl_employee and tbl_employee_training tables together. The `where` clause provides further filtering that limits the result set to those records in the tbl_employee table that have a matching EmployeeID entry in a tbl_employee_training record. (The term *record* is synonymous with the term *row*.) To run this query, start the SQL command utility with the following command:

```
sqlcmd -S .\sqlexpress -W
```

The `-W` switch removes the trailing spaces from each field so the query results fit on the screen.

Figure 20-28 shows the results of running this query in the SQL command utility against our freshly-loaded test data.

Referring to Figure 20-28 — note that the number of results equals the number of training records contained in the tbl_employee_training table.

Now, suppose you wanted to find only those employees who've attended Project Management Professional training and sort the results by the employee's last name. The query for that question would look like this:

```
select firstname, lastname
from tbl_employee, tbl_employee_training
where (tbl_employee.EmployeeID = tbl_employee_training.EmployeeID) AND
       (title = 'Project Management Professional')
order by lastname
go
```

In this example, the `where` clause uses the `AND` operator to provide the required record filtering. The results of running this query are shown in Figure 20-29.

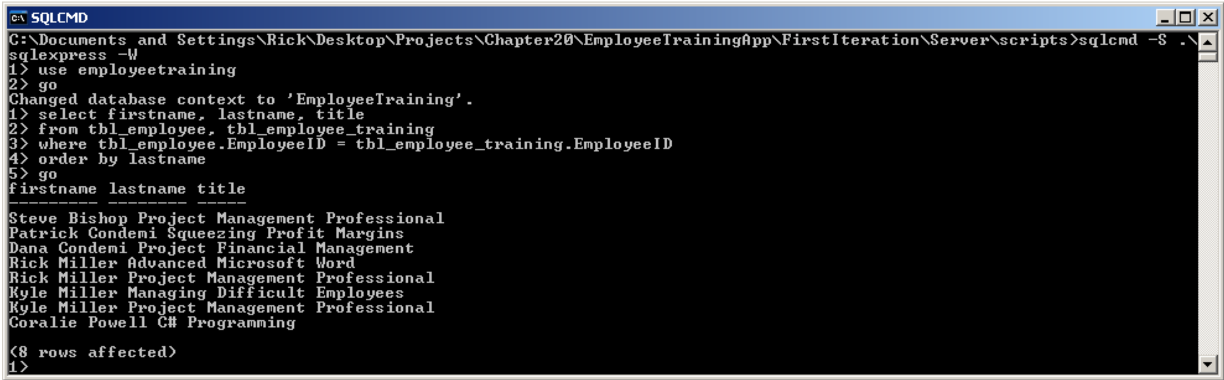


Figure 20-28: Results of Running the Previous SQL Query

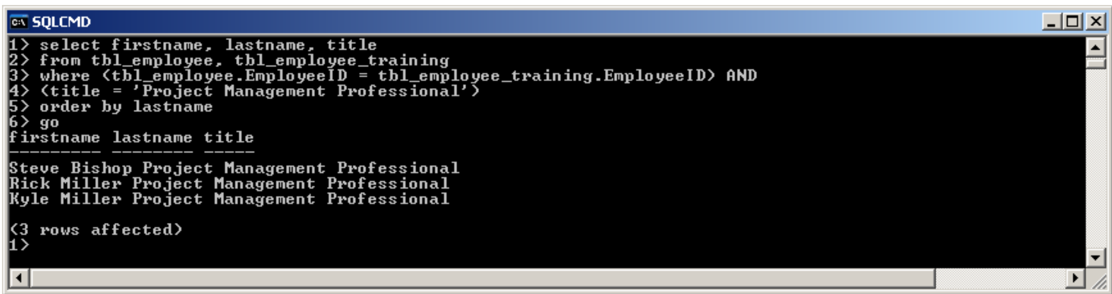


Figure 20-29: Results of Running the Previous SQL Query

Testing The Cascade Delete Constraint

If you'll return to Example 20.7 you'll see on line 37 that the foreign key constraint specifies that when a row from the tbl_employee table is deleted, all related records in the tbl_employee_training table will also be deleted. Let's test the cascade delete mechanism now by deleting the employee Rick Miller from the database. The SQL `delete` statement would look something like this:

```
delete from tbl_employee
where employeeid = 'E4F786EC-D8FC-472A-9E8C-4DDE307ABEC8'
go
```

After you execute this `delete` statement, you'll want to check to be sure the related training records were in fact deleted. You can do that with the following query:

```
select * from tbl_employee_training
```

The results of executing these two statement are shown in Figure 20-30.

Quick Review

The `select` statement can be used to construct complex queries involving multiple related tables. One table is joined to another to form a temporary table. There are many different types of join operations, but the most common one is an *inner join*, which is the default join condition provided by Microsoft SQL Server.

Inner joins are made possible through the use of *foreign keys*. A foreign key is a column in a table that contains a value that is used as a primary key in another table. A table can be related to many other tables by including multiple foreign keys. Specify a foreign key by adding a *foreign key constraint* to a particular table using the `alter` command.

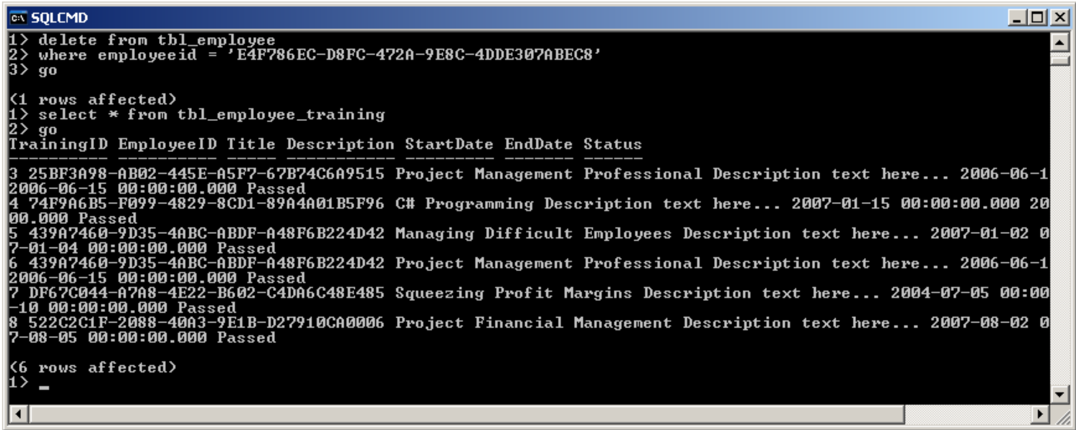


Figure 20-30: Results of Executing a Cascade Delete and Checking the Results

The Server Application

Now that you have a better understanding of relational databases and Structured Query Language, it's time to move on to building the employee training application. The best way to approach the design and development of a complex application is through the use of development iterations. In this section I will step through the development of the employee training server application. As is the case with any complex development project, the best way to start is to get organized. I recommend adopting a project folder structure that mirrors the application layers or tiers.

Project Folder Organization

Figure 20-31 shows how I've arranged my server application project folders.

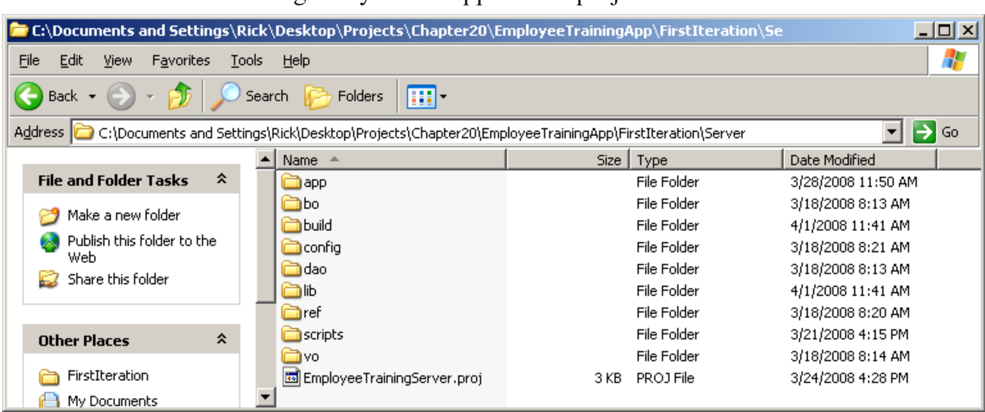


Figure 20-31: Employee Training Project Folder Arrangement

Referring to Figure 20-31 — the structure of my project folder mirrors that of the application layers contained within my application plus several more folders to hold different types of project artifacts. You'll also notice that there is an EmployeeTrainingServer.proj file at the bottom of the list. This is an MSBuild project file that is used to

manage and build the project. I'll explain the use of the MSBuild project file in a moment. Table 20-2 lists and describes the purpose and contents of each of the folders shown above.

Folder Name	Contents
app	Contains source code files for the main server application, remote object interface, and remote object.
bo	Contains source code files for business objects.
build	Stores the resultant application build files. This includes dlls, the config file, and the server.exe file. Most of the files in this folder are copied from other project folders.
config	Contains the master copy of the server config file.
dao	Contains the source code files for the data access objects.
lib	Stores application dlls after they have been built. Other parts of the project will depend on the files stored in the lib directory.
ref	Stores dlls and other third-party libraries. These are libraries the server application depends on to build and run but are not built by the application build process.
scripts	Contains database scripts.
vo	Contains the source code files for the value objects.

Table 20-2: Project Folder Descriptions

Using Microsoft Build To Manage And Build the Project

Due to the complexity of the employee training server application, it would be difficult at best to compile the project files from the command line using only the `csc` compiler tool. The Microsoft Build tool (MSBuild) enables you to build complex projects with the help of project files. Example 20.9 gives the code for the EmployeeTraining-Server.proj file. You will find the syntax of this file somewhat confusing at first, however, keep studying it until you understand what's going on. Knowing how to use MSBuild will save you a ton of time.

20.9 EmployeeTrainingServer.proj

```
<Project DefaultTargets="CompileVO"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

   <PropertyGroup>
     <IncludeDebugInformation>false</IncludeDebugInformation>
     <BuildDir>build</BuildDir>
     <LibDir>lib</LibDir>
     <AppDir>app</AppDir>
     <RefDir>ref</RefDir>
     <ConfigDir>config</ConfigDir>
   </PropertyGroup>

    <ItemGroup>
      <DAO Include="dao\**\*.cs" />
      <BO Include="bo\**\*.cs" />
      <VO Include="vo\**\*.cs" />
      <APP Include="app\**\*.cs" />
      <LIB Include="lib\**\*.dll" />
      <REF Include="ref\**\*.dll" />
      <CONFIG Include="config\**\*.config" />
      <EXE Include="app\**\*.exe" />
    </ItemGroup>

    <Target Name="MakeDirs">
      <MakeDir Directories="$(BuildDir)" />
      <MakeDir Directories="$(LibDir)" />
    </Target>

    <Target Name="RemoveDirs">
      <RemoveDir Directories="$(BuildDir)" />
      <RemoveDir Directories="$(LibDir)" />
    </Target>
```



```

    <Target Name="Clean"
            DependsOnTargets="RemoveDirs;MakeDirs">
    </Target>

     <Target Name="CopyFiles">
       <Copy
         SourceFiles="@(CONFIG);@(LIB);@(REF)"
         DestinationFolder="$(BuildDir)" />
    </Target>

    <Target Name="CompileVO"
            Inputs="@(VO)"
            Outputs="$(LibDir)\VOLib.dll">
      <Csc Sources="@(VO)"
          TargetType="library"
          References="@(REF);@(LIB)"
          OutputAssembly="$(LibDir)\VOLib.dll">
      </Csc>
   </Target>

    <Target Name="CompileDAO"
            Inputs="@(DAO)"
            Outputs="$(LibDir)\DAOLib.dll"
            DependsOnTargets="CompileVO">
       <Csc Sources="@(DAO)"
          TargetType="library"
          References="@(REF);@(LIB)"
          WarningLevel="0"
          OutputAssembly="$(LibDir)\DAOLib.dll">
      </Csc>
    </Target>

    <Target Name="CompileBO"
            Inputs="@(BO)"
            Outputs="$(LibDir)\BOLib.dll"
            DependsOnTargets="CompileDAO">
      <Csc Sources="@(BO)"
          TargetType="library"
          References="@(REF);@(LIB)"
          WarningLevel="0"
          OutputAssembly="$(LibDir)\BOLib.dll">
      </Csc>
    </Target>

    <Target Name="CompileApp"
            Inputs="@(APP)"
            Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
            DependsOnTargets="CompileDAO">
      <Csc Sources="@(APP)"
           TargetType="exe"
           References="@(REF);@(LIB)"
           OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
      </Csc>
    </Target>

    <Target Name="CompileAll">
      <Csc Sources="@(VO);@(DAO);@(BO);@(APP)"
           TargetType="exe"
           References="@(REF);@(LIB)"
           OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
      </Csc>
    </Target>

    <Target Name="Run"
            DependsOnTargets="CompileApp;CopyFiles">
      <Exec Command="$(MSBuildProjectName).exe"
            WorkingDirectory="$(BuildDir)" />
    </Target>

</Project>
```

Referring to Example 20.9 — the EmployeeTrainingServer.proj file contains a project specification between a pair of XML <project></project> tags. Within the project tags there appears a PropertyGroup specification, an ItemGroup specification, and several Targets.

The PropertyGroup specification appears between the <PropertyGroup></PropertyGroup> tags and defines a list of properties used within the project. Properties within the project are referenced via the $(*PropertyName*) notation.

Most of the properties defined are project folder names. For example, on line 6, the <BuildDir> property is defined as the *build* directory.

The ItemGroup specification appears between the <ItemGroup></ItemGroup> tags and defines a list of project artifacts. An item within the project is referenced with the @(*ItemName*) notation. Items defined in this project include source files in various directories (.cs), library files (.dlls), config files, and executable files (.exe). For example, the DAO item defined on line 14 includes all the C# source files found in the dao directory and all its subdirectories.

The remainder of the EmployeeTrainingServer.proj file contains target definitions. A *target* is an action the MSBuild tool will perform and includes a set of one or more tasks. A target definition appears between the <Target></Target> tags. Targets can be stand-alone or they can depend on other targets. For example, the Clean target defined on line 34 depends on the RemoveDirs and MakeDirs targets. In other words, running the Clean target will also run the RemoveDirs and MakeDirs targets.

MSBuild projects have a default target. For example, on line 1 you see the default target for the EmployeeTrainingServer.proj is the CompileVO project.

Let's take a look at one of the more complex targets. The CompileApp target definition begins on line 78. Its inputs include all the source files in the app directory, as specified in the APP item and referenced with @(APP). Its output is an executable file written to the build directory, as specified by the BuildDir property and referenced with $(BuildDir). The CompileApp target depends on the CompileDAO target. (**Note:** This dependency will change once we move into the second iteration of the server application development.) The CompileApp target contains one compile task as is specified with the <Csc></Csc> tags. The Csc task calls the C# compiler tool and compiles all the source files found in the app directory, builds the .exe file, and writes it to the build directory. The Csc task references the libraries found in the lib and ref directories.

(**Note:** This version of the project file will change slightly as the project evolves.) As I mentioned above, the CompileApp target currently depends on the CompileDAO target. This dependency will change to the CompileBO target once we start working on the application's business objects. Also note that the CompileVO, CompileBO, and CompileDAO targets all produce dlls. These dlls are written to the lib directory.

I'll show you how to run the build using MSBuild as soon as we get some source code to compile. So, let's start on the first iteration of the employee training server application.

First Iteration

Let's see, where do we stand? The database is up and running. We have database scripts that can be used to drop and create the database, the required tables, and test data. I think a good overall objective for the first iteration of any development project is to identify, design, and code the high-risk areas. (*i.e.,* Solve the most difficult problems first.) For this project, the most difficult aspect is the DAO layer and the insertion and retrieval of an employee's data and their picture. Also, with multitier projects like this one, it's a good idea to code from the database out, meaning again that the DAO layer deserves our attention right from the start. Given this assessment, the objectives for the first development iteration are listed in Table 20-3.

Check-Off	Design Consideration	Design Decision
	DAO layer	Create a data access object (a C# class) for the employee table. Focus on the insertion and retrieval of employee data including the employee's picture. The EmployeeDAO class will need a connection to the database. This is a good use for a BaseDAO class.
	Value objects	Value objects represent entities within the application that are passed between tiers. A good place to start would be to create an EmployeeVO that contains all an employee's data. In past chapters we've already created a Person class that has most of the properties required by the EmployeeVO class. You can let the Person class serve as the base class for the EmployeeVO. For consistency we'll rename the Person class to be PersonVO.

Table 20-3: Employee Training Server Application — First Iteration Design Considerations & Decisions

Check-Off	Design Consideration	Design Decision
	Enterprise Library Data Access Application Block	The Enterprise Library Data Access Application Block provides a DatabaseFactory class. You'll need to create an application configuration file that provides the required database connection. The name of the configuration file will be: `EmployeeTrainingServer.exe.config` Place this file in the project's config directory. You'll also need to copy and paste the following enterprise library dlls into the project's ref directory: Microsoft.Practices.EnterpriseLibrary.Common.dll Microsoft.Practices.EnterpriseLibrary.Data.dll Microsoft.Practices.ServiceLocation.dll Microsoft.Practices.Unity.dll Microsoft.Practices.Unity.Interception.dll
	Test application	You'll need to write a small application that tests the EmployeeDAO. The application should let you select an image to use for the employee's picture so it will be a GUI application. It doesn't need to be fancy as it will be thrown away. The name of the application source file will be: `EmployeeTrainingServer.cs` Create this file in the project's app directory.

Table 20-3: Employee Training Server Application — First Iteration Design Considerations & Decisions

Referring to Table 20-3 — this looks like enough work for now. Although this development cycle will yield only five source files: EmployeeTrainingServer.cs, BaseDAO.cs, EmployeeDAO.cs, PersonVO.cs, and EmployeeVO.cs, it exercises a major portion of the architecture and forces you to deal with the most complex issues you'll face during the development of this project, and that is coding up the DAO layer. I must remind you before proceeding that design decisions made early on a complex project like this one will most certainly change before the project ends. This is the natural state of affairs in software development. If the application architecture is flexible enough to be changed without too much pain then the design is sound.

Coding The EmployeeVO And EmployeeDAO

Figure 20-32 gives the UML diagram for the EmployeeVO and EmployeeDAO classes.

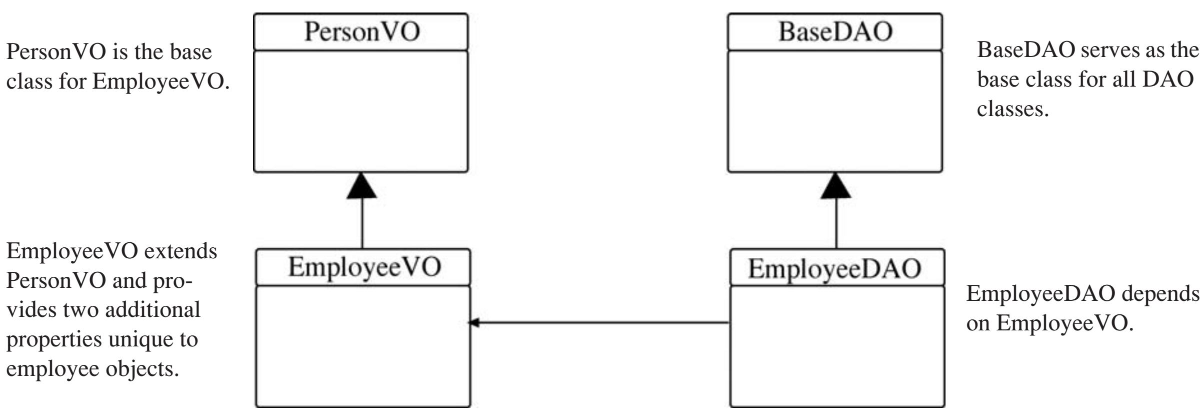


Figure 20-32: EmployeeVO and EmployeeDAO Class Diagram

Referring to Figure 20-32 — the EmployeeDAO extends the BaseDAO class and has a dependency association on the EmployeeVO class. The EmployeeVO class extends PersonVO. Since the EmployeeDAO depends on the EmployeeVO class, you must code it first. Examples 20.10 and 20.11 give the code for these classes.

20.10 PersonVO.cs

```
using System;

```



```
3   namespace EmployeeTraining.VO {
4     [Serializable]
5     public class PersonVO {
6
7     //enumeration
8     public enum Sex {MALE, FEMALE};
9
10    // private instance fields
11    private String   _firstName;
12    private String   _middleName;
13    private String   _lastName;
14    private Sex      _gender;
15    private DateTime _birthday;
16
17    //default constructor
18    public PersonVO(){}
19
20    public PersonVO(String firstName, String middleName, String lastName,
21                  Sex gender, DateTime birthday){
22       FirstName = firstName;
23       MiddleName = middleName;
24       LastName = lastName;
25       Gender = gender;
26       BirthDay = birthday;
27    }
28
29    // public properties
30    public String FirstName {
31      get { return _firstName; }
32      set { _firstName = value; }
33    }
34
35    public String MiddleName {
36      get { return _middleName; }
37      set { _middleName = value; }
38    }
39
40    public String LastName {
41      get { return _lastName; }
42      set { _lastName = value; }
43    }
44
45    public Sex Gender {
46      get { return _gender; }
47      set { _gender = value; }
48    }
49
50    public DateTime BirthDay {
51      get { return _birthday; }
52      set { _birthday = value; }
53    }
54
55    public int Age {
56       get {
57       int years = DateTime.Now.Year - _birthday.Year;
58         int adjustment = 0;
59       if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60         adjustment = 1;
61       }
62       return years - adjustment;
63     }
64     }
65
66    public String FullName {
67      get { return FirstName + " " + MiddleName + " " + LastName; }
68    }
69
70    public String FullNameAndAge {
71      get { return FullName + " " + Age; }
72    }
73
74    public override String ToString(){
75      return FullName + " is a " + Gender + " who is " + Age + " years old.";
76    }
77
78    } // end PersonVO class
79  } // end namespace
```

Referring to Example 20.10 — the PersonVO class belongs to the EmployeeTraining.VO namespace, as line 3 indicates.

20.11 EmployeeVO.cs

```
using System;
using System.Drawing;

namespace EmployeeTraining.VO {
[Serializable]
  public class EmployeeVO : PersonVO {

  // private instance fields
  private Guid    _employeeID;
  private Image   _picture;

  //default constructor
  public EmployeeVO(){}

  public EmployeeVO(Guid employeeid, String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday):base(firstName, middleName, lastName, gender, birthday){
     EmployeeID = employeeid;
  }

  // public properties
  public Guid EmployeeID {
    get {  return _employeeID;  }
    set {  _employeeID = value; }
  }

  public Image Picture {
    get {  return _picture;  }
    set {  _picture = value; }
  }

  public override String ToString(){
    return (EmployeeID + " " + base.ToString());
  }
} // end EmployeeVO class
} // end namespace
```

Referring to Example 20.11 — the EmployeeVO class is quite simple because most of the heavy lifting is done by the PersonVO class. This class adds two additional properties: EmployeeID, which is of type System.Guid (Globally Unique Identifier), and Picture, which is of type System.Drawing.Image. Note that both the PersonVO and EmployeeVO classes are tagged with the Serializable attribute.

To compile these classes with the MSBuild project file, make sure both classes are located in the project's vo directory, change to the server directory, and run the project file with the following command:

```
msbuild /target:compilevo
```

If you get compile errors, edit the files accordingly and run the build again. Eventually, your output should look similar to that shown in Figure 20-33.

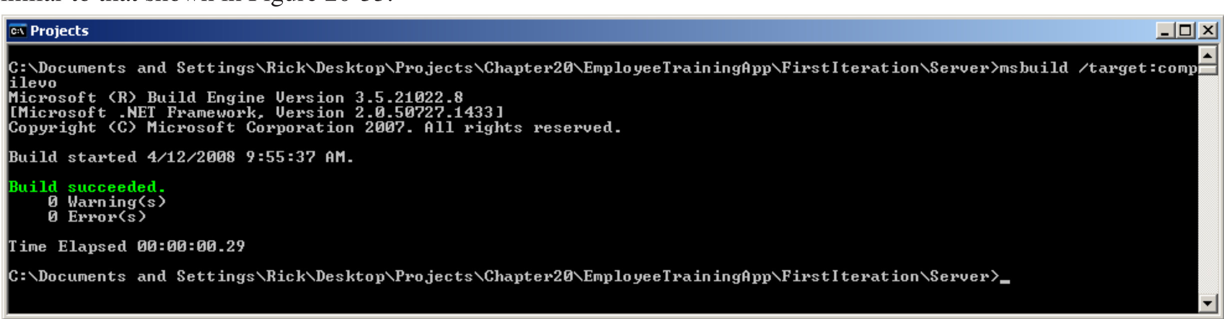


Figure 20-33: Results of Running the CompileVO Target using the MSBuild Utility

At this point you should check to ensure the build did in fact write the VOLib.dll to the lib directory. If not, check the validity of the EmployeeTrainingServer.proj file and make sure your project folder names match those of the properties defined within the project file. Then try and try again until you get this build target to work correctly.

Now, if you edit either the PersonVO or EmployeeVO source files and run the compilevo target again without running the clean target, you'll get the type conflict warnings shown in Figure 20-34.

You can safely ignore these warnings. What's happening here is that when the <Csc> task executes it references the VOLib.dll, which now resides in the lib directory. This dll contains the definition for the PersonVO. The warning message states that the compiler is using the definition found in PersonVO.cs instead, which is perfectly fine.

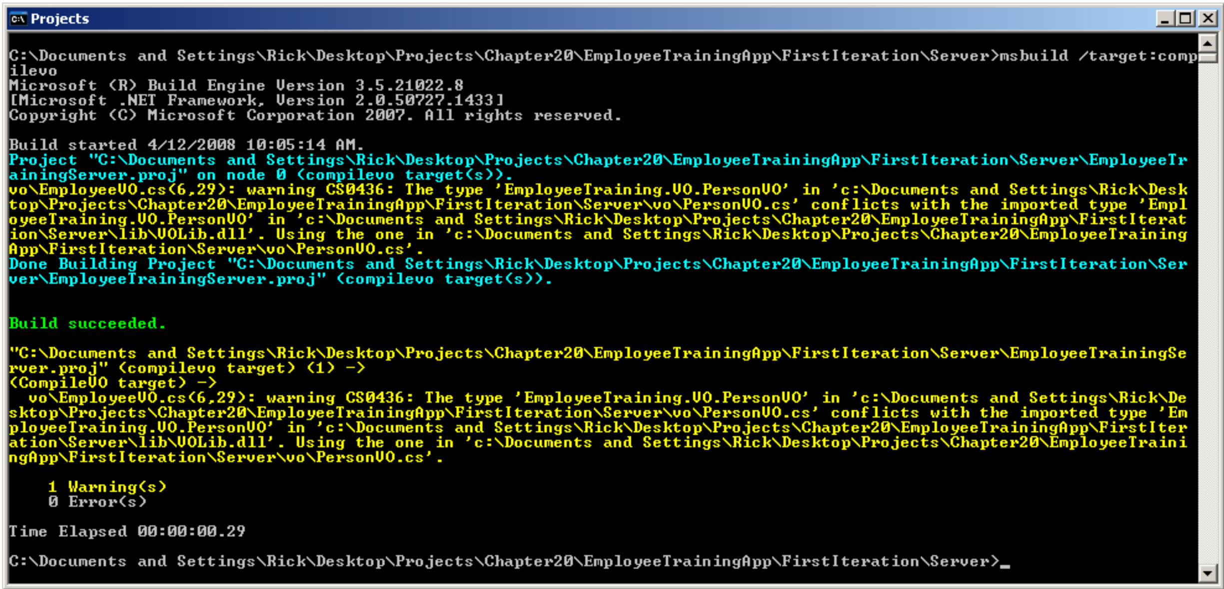


Figure 20-34: Build Warnings From Conflicting Type Declarations

Now that the EmployeeVO is coded up, you can move on to the EmployeeDAO. You might want to write a short application to test the EmployeeVO but I'm skipping that step in this chapter. In a production environment, you'd use a testing framework like NUnit to write unit tests that thoroughly exercise the classes you create in your application. (Unfortunately, I don't have the space to cover the use of NUnit in this book but I recommend you explore its capabilities on your own when you have a chance.)

Example 20-12 gives the code for the BaseDAO class.

20.12 BaseDAO.cs

```
1   using System;
2   using System.Data;
3   using System.Configuration;
4
5   using Microsoft.Practices.EnterpriseLibrary.Data;
6
7   namespace EmployeeTraining.DAO {
8     public class BaseDAO {
9       private Database _database;
10
11      protected Database DataBase {
12        get {
13          if(_database == null){
14            try {
15              _database = DatabaseFactory.CreateDatabase();
16            }catch(ConfigurationException ce){
17              Console.WriteLine(ce);
18            }
19          }
20          return _database;
21        }
22      }
23
24      protected void CloseReader(IDataReader reader){
25        if(reader != null){
26          try{
27            reader.Close();
28          }catch(Exception e){
29            Console.WriteLine(e);
30          }
31        }
32      }
33
34    } // end BaseDAO class definition
35  } // end namespace
```

Referring to Example 20.12 — the purpose of this class is to make available to its subclasses a Database object via its DataBase property. This class implements the Singleton software design pattern. The DataBase property definition begins on line 11. If the _database field is null, a call is made to the DatabaseFactory.CreateDatabase() method

to create the Database object. If the _database field is not null, the property simply returns the existing reference. This class also provides a CloseReader() method used by its subclasses to close the IDatabaseReader object.

Example 20-13 lists the 1st iteration implementation of the EmployeeDAO class. For this iteration I focused on the insertion and retrieval of EmployeeVO object data into the database. In the EmployeeDAO class you'll find all the SQL code required to create, read, update, and delete (CRUD) employee database records, although in this initial version of the code I've only implemented the create (*i.e.,* insert) and read (*i.e.,* get) operations.

20.13 EmployeeDAO.cs (1st Iteration)

```
using System;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Data.Sql;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using EmployeeTraining.VO;

using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Microsoft.Practices.EnterpriseLibrary.Data.Sql;

namespace EmployeeTraining.DAO {
  public class EmployeeDAO : BaseDAO {

   private bool debug = true;

    //List of column identifiers used in perpared statements
    private const String EMPLOYEE_ID = "@employee_id";
    private const String FIRST_NAME = "@first_name";
    private const String MIDDLE_NAME = "@middle_name";
    private const String LAST_NAME = "@last_name";
    private const String BIRTHDAY = "@birthday";
    private const String GENDER = "@gender";
    private const String PICTURE = "@picture";

    private const String SELECT_ALL_COLUMNS =
      "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";

    private const String SELECT_ALL_EMPLOYEES =
      SELECT_ALL_COLUMNS +
      "FROM tbl_employee ";

    private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
      SELECT_ALL_EMPLOYEES +
      "WHERE employeeid = " + EMPLOYEE_ID;


    private const String INSERT_EMPLOYEE =
      "INSERT INTO tbl_employee " +
          "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
       "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
                    BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";


 /*************************************
    Returns a List<EmployeeVO> object
 *************************************/
    public List<EmployeeVO> GetAllEmployees(){
      DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
      return this.GetEmployeeList(command);
    }

 /***********************************************
    Returns an EmployeeVO object given a valid employeeid
 ************************************************/
    public EmployeeVO GetEmployee(Guid employeeid){
      DbCommand command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
      DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
      return this.GetEmployee(command);
    }

 /*****************************************************
    Inserts an employee given a fully-populated EmployeeVO object
 ******************************************************/
    public EmployeeVO InsertEmployee(EmployeeVO employee){
```



```
      try{
        employee.EmployeeID = Guid.NewGuid();
        DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
        DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
        DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
        DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
        DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
        switch(employee.Gender){
          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
               break;
          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
               break;
        }

        if(employee.Picture != null){
         if(debug){ Console.WriteLine("Inserting picture!"); }
          MemoryStream ms = new MemoryStream();
          employee.Picture.Save(ms, ImageFormat.Tiff);
          byte[ ] byte_array = ms.ToArray();
          if(debug){
            for(int i=0; i<byte_array.Length; i++){
              Console.Write(byte_array[ i] );
            }
          } // end if debug
         DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
         if(debug){ Console.WriteLine("Picture inserted, I think!"); }
        }

        DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetEmployee(employee.EmployeeID);
    }

 /***********************************************************
    Private utility method that executes the given DbCommand
    and returns a fully-populated EmployeeVO object
 ************************************************************/
    private EmployeeVO GetEmployee(DbCommand command){
      EmployeeVO empVO = null;
      IDataReader reader = null;
      try {
        reader = DataBase.ExecuteReader(command);
        if(reader.Read()){
          empVO = this.FillInEmployeeVO(reader);
        }
      } catch(Exception e){
        Console.WriteLine(e);
      } finally {
        base.CloseReader(reader);
      }
      return empVO;
    }

 /*****************************************************
    GetEmployeeList() - returns a List<EmployeeVO> object
 ******************************************************/
    private List<EmployeeVO> GetEmployeeList(DbCommand command){
      IDataReader reader = null;
      List<EmployeeVO> employee_list = new List<EmployeeVO>();
      try{
        reader = DataBase.ExecuteReader(command);
        while(reader.Read()){
          EmployeeVO empVO = this.FillInEmployeeVO(reader);
          employee_list.Add(empVO);
        }
      } catch(Exception e){
        Console.WriteLine(e);
      } finally{
        base.CloseReader(reader);
      }
      return employee_list;
    }

 /*****************************************************************
    Private utility method that populates an EmployeeVO object from
    data read from the IDataReader object
 ******************************************************************/
    private EmployeeVO FillInEmployeeVO(IDataReader reader){
```



```
152       EmployeeVO empVO = new EmployeeVO();
153       empVO.EmployeeID = reader.GetGuid(0);
154       empVO.FirstName = reader.GetString(1);
155       empVO.MiddleName = reader.GetString(2);
156       empVO.LastName = reader.GetString(3);
157       empVO.BirthDay = reader.GetDateTime(4);
158       String gender = reader.GetString(5);
159       switch(gender){
160         case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
161                    break;
162         case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
163                    break;
164       }
165       if(!reader.IsDBNull(6)){
166         int buffersize = 5000;
167         int startindex = 0;
168         Byte[] byte_array = new Byte[buffersize];
169         MemoryStream ms = new MemoryStream();
170         long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
171         while(retval > 0){
172           ms.Write(byte_array, 0, byte_array.Length);
173           startindex += buffersize;
174           retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
175          }
176         empVO.Picture = new Bitmap(ms);
177       }
178       return empVO;
179     }
180   } // end EmployeeDAO definition
181 } // end namespace
```

Referring to Example 20.13 — lines 23 through 29 define SQL command parameter string constants representing each column in the tbl_employee table. Note that these are not the same as verbatim strings. The difference lies in the placement of the @ symbol. This is a verbatim string:

`@"this is a verbatim string"`

This is an SQL command parameter string:

`"@this is an SQL parameter string"`

The SQL command parameter string constants are then used to create SQL query string constants, which are used later to create *prepared statements*. Let's see how this is done by tracing the execution of the InsertEmployee() method, which begins on line 70.

The InsertEmployee() method takes a populated EmployeeVO object as an argument. Since this is a new employee, the incoming EmployeeVO object lacks a valid EmployeeID, so the first thing that must be done is to make a call to the Guid.NewGuid() method to generate a valid globally unique identifier. This Guid value will become the employee's primary key.

On line 73, the BaseDAO's DataBase property (which is a Database object) is used to create a DbCommand object with a call to its GetSqlStringCommand() method. The argument to this method call is the INSERT_EMPLOYEE SQL string constant, which is defined on line 43. Refer now to line 43 to see how the SQL command parameters are used to formulate the INSERT_EMPLOYEE query string. Note the correspondence between each SQL command parameter included in the INSERT_EMPLOYEE string and lines 74 through 84 where the DataBase.AddInParameter() method is called to set the value of each SQL command parameter.

The `switch` statement beginning on line 79 checks the value of the incoming EmployeVO.Gender property and sets the GENDER command parameter to the corresponding valid one-character value required by the tbl_employee.Gender column.

The employee picture insertion code begins on line 86. If the incoming EmployeeVO.Picture property is null, I skip the insertion. This is valid because the tbl_employee.Picture column is allowed to contain null values. If the EmployeeVO.Picture property is not null then it's converted into a byte array (byte[]). To do this I save the Picture data to a MemoryStream and then call the MemoryStream's ToArray() method, which returns the required byte array. I've also included some debugging code that allows me to trace the insertion of the picture data if the class constant *debug* is true. (lines 87, 91, and 97)

When all the command parameters have been set, I execute the DbCommand by calling the DataBase.Execute–NonQuery() method.

SQL Command Parameters And Prepared Statements: Generalized Steps

So, in a nutshell, here are the generalized steps to using SQL command parameters and prepared statements:

Step 1: Define the required SQL command parameters. There is usually a one-to-one correspondence between a command parameter and a column in the targeted database table.

Step 2: Create an SQL command string using the previously defined command parameters.

Step 3: Create a DbCommand object by calling the Database.GetSqlCommandString() method passing in as an argument the SQL command string.

Step 4: Set each command parameter value with a call to Database.AddInParameter() method.

Step 5: Execute the DbCommand with a call to ExecuteNonQuery() (or ExecuteReader() or ExecuteScalar() methods.)

DbType Enumeration Values And .NET Type Mapping

Refer for a moment to line 74 of the EmployeeDAO class. The Database.AddInParameter() method takes four arguments. These include a DbCommand reference, an SQL command parameter string, a DbType, and the value you want to use to set the SQL command parameter. The DbType enumeration is located in the System.Data namespace and defines a list of database types available to a .NET data provider. The type of the value you want to set the SQL command parameter to must correspond to the appropriate DbType, which must correspond to the MS SQL Server database type supported by the targeted database table column. Table 20-4 offers a mapping table between these three types and the corresponding IDataReader methods.

.NET Type	DbType	SQL Server Type	IDataReader Methods
String	AnsiString	varchar	GetString()
byte byte[]	Binary	varbinary	GetByte(), GetBytes()
byte	Byte	binary, varbinary	GetByte()
bool	Boolean	bit	GetBoolean()
decimal	Currency	money smallmoney	GetDecimal()
DateTime	Date	datetime smalldatetime	GetDateTime()
DateTime	DateTime	datetime smalldatetime	GetDateTime()
decimal	Decimal	decimal	GetDecimal()
double	Double	float	GetDouble()
Guid	Guid	uniqueidentifier	GetGuid()
short	Int16	smallint	GetInt16()
int	Int32	int	GetInt32()
long	Int64	bigint	GetInt64()
Object	Object	varbinary	GetValue()
sbyte	SByte	binary	GetBinary()

Table 20-4: .NET to DbType to SQL Server Type to IDataReader Method Mapping

.NET Type	DbType	SQL Server Type	IDataReader Methods
float	Single	float real	GetFloat()
String char[]	String	char, varchar, text nchar, nvarchar	GetChar(), GetChars() GetString()
DateTime	Time	datetime	GetDateTime()
ushort	UInt16		
uint	UInt32		
ulong	UInt64		
	VarNumeric		
	AnsiStringFixedLength		
XMLDocument	Xml	xml	
DateTime	DateTime2		
DateTime	DateTimeOffset		

Table 20-4: .NET to DbType to SQL Server Type to IDataReader Method Mapping

Referring to Table 20-4 — note that there is not a one-to-one correspondence between all .NET, DbType, and SQL Server types.

Application Configuration File

Example 20.14 gives the configuration file for the first iteration of the Employee Training application. You can create this file with the Enterprise Library Configuration tool, which was covered earlier in the chapter.

20.14 EmployeeTrainingServer.exe.config (1st iteration version)

```
<configuration>
  <configSections>
     <section name="dataConfiguration"
              type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
                   Microsoft.Practices.EnterpriseLibrary.Data,
                   Version=5.0.414.0,
                   Culture=neutral,
                   PublicKeyToken=31bf3856ad364e35"
                   requirePermission="true" />
    </configSections>
  <dataConfiguration defaultDatabase="Connection String" />
   <connectionStrings>
      <add name="Connection String" connectionString="Data Source=(local)\SQLEXPRESS;
                Initial Catalog=EmployeeTraining;Integrated Security=True"
                providerName="System.Data.SqlClient" />
  </connectionStrings>
  <system.runtime.remoting>
    <application>
      <service>
       <wellknown mode="Singleton" type="TestClassTwo, TestClassTwo" objectUri="TestClass" />
      </service>
      <channels>
        <channel ref="tcp" port="8080" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Referring to Example 20-14 — this version provides the necessary database connection information required for the DatabaseFactory class. Later, I will add to this file a remoting section to configure the remote object, but for now it’s fine the way it stands.

Creating Test Application

All that's left now is to write a brief test application that can be used to create and retrieve employee objects and test the DAO layer. Example 20-15 gives the code for a GUI application that provides a PictureBox and several buttons. The primary goal of this test application is to allow the selection and insertion of an employee picture. I do not particularly care about creating different employees per se, so there are no text boxes with which to enter employee data like an employee's first name, last name, etc. I instead create the same employee, Rick Miller.

20.15 EmployeeTrainingServer.cs (Throw away test code)

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Imaging;
using System.Collections.Generic;
using EmployeeTraining.DAO;
using EmployeeTraining.VO;

public class EmployeeTrainingServer : Form  {

  private PictureBox _picturebox;
  private TableLayoutPanel _tablepanel;
  private FlowLayoutPanel _flowpanel;
  private Button _button1;
  private Button _button2;
  private Button _button3;
  private Button _button4;
  private Button _button5;
  private EmployeeVO _emp_vo;
  private List<EmployeeVO> _list;
  private int _next_employee = 0;
  private OpenFileDialog _dialog;

  public EmployeeTrainingServer(){
    this.InitializeComponent();
    Application.Run(this);
  }

  private void InitializeComponent(){
     this.SuspendLayout();
    _tablepanel = new TableLayoutPanel();
    _flowpanel = new FlowLayoutPanel();
    _tablepanel.SuspendLayout();
    _tablepanel.RowCount = 1;
    _tablepanel.ColumnCount = 2;
    _tablepanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
    _tablepanel.Dock = DockStyle.Left;
    _tablepanel.Width = 600;

    _picturebox = new PictureBox();
    _picturebox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;

    _button1 = new Button();
    _button1.Text = "Create";
    _button1.Click += this.CreateEmployee;
    _button1.Enabled = false;

    _button2 = new Button();
    _button2.Text = "Load";
    _button2.Click += this.LoadEmployee;
    _button2.Enabled = false;

    _button3 = new Button();
    _button3.Text = "Find Picture";
    _button3.Click += this.ShowOpenFileDialog;

    _button4 = new Button();
    _button4.Text = "Get All Employees";
    _button4.AutoSize = true;
    _button4.Click += this.GetAllEmployees;

    _button5 = new Button();
    _button5.Text = "Next";
    _button5.Click += this.NextEmployee;
    _button5.Enabled = false;

    _tablepanel.Controls.Add(_picturebox);
    _flowpanel.Controls.Add(_button1);
    _flowpanel.Controls.Add(_button2);
```



```
      _flowpanel.Controls.Add(_button3);
      _flowpanel.Controls.Add(_button4);
      _flowpanel.Controls.Add(_button5);
      _tablepanel.Controls.Add(_flowpanel);

      this.Controls.Add(_tablepanel);
      this.Width = _tablepanel.Width;
      this.Height = 300;
      _tablepanel.ResumeLayout();
      this.ResumeLayout();
      _dialog = new OpenFileDialog();
      _dialog.FileOk += this.LoadPicture;
    }

    public void ShowOpenFileDialog(Object sender, EventArgs e){
      _dialog.ShowDialog();
    }

    public void LoadPicture(Object sender, EventArgs e){
       String filename = _dialog.FileName;
      _picturebox.Image = new Bitmap(filename);
      this.AdjustPicturebox();
      _button1.Enabled = true;
    }

    public void CreateEmployee(Object sender, EventArgs e){
      EmployeeVO vo = new EmployeeVO();
      vo.FirstName = "Rick";
      vo.MiddleName = "Warren";
      vo.LastName = "Miller";
      vo.Gender = EmployeeVO.Sex.MALE;
      vo.BirthDay = new DateTime(1961, 2, 4);
      vo.Picture = _picturebox.Image;

      EmployeeDAO dao = new EmployeeDAO();
      _emp_vo = dao.InsertEmployee(vo);
      _picturebox.Image = null;
      _button2.Enabled = true;
      _button1.Enabled = false;
    }

    public void LoadEmployee(Object sender, EventArgs e){
      EmployeeDAO dao = new EmployeeDAO();
      _emp_vo.Picture = null;
      _emp_vo = dao.GetEmployee(_emp_vo.EmployeeID);
      _picturebox.Image = _emp_vo.Picture;
    }

    public void GetAllEmployees(Object sender, EventArgs e){
      EmployeeDAO dao = new EmployeeDAO();
      _list = dao.GetAllEmployees();
      foreach(EmployeeVO emp in _list){
        Console.WriteLine(emp);
      }
      _button5.Enabled = true;
    }

    public void NextEmployee(Object sender, EventArgs e){
      Console.WriteLine(_next_employee);
      if(_next_employee >= _list.Count){
        _next_employee = 0;
      }
      Console.WriteLine(_next_employee);
      Console.WriteLine(_list[_next_employee]);
      _picturebox.Image = _list[_next_employee++].Picture;
      if(_picturebox.Image != null){
        this.AdjustPicturebox();
     }
    }

    private void AdjustPicturebox(){
      this.SuspendLayout();
      _tablepanel.SuspendLayout();
      _picturebox.Width = _picturebox.Image.Width;
      _picturebox.Height = _picturebox.Image.Height;
      _tablepanel.Width = _picturebox.Image.Width + 300;
      this.Width = _tablepanel.Width;
      _tablepanel.ResumeLayout();
      this.ResumeLayout();
    }

```



```
151   [STAThread]
152   public static void Main(){
153     new EmployeeTrainingServer();
154   }
155 }
```

Referring to Example 20-15 — this application displays a form that contains a TableLayoutPanel. The TableLayoutPanel contains a PictureBox and five buttons. To run this application, make sure you're in the directory that contains the EmployeeTrainingServer.proj file and enter the following MSBuild command on the command line:

```
msbuild /target:run
```

The startup window will look similar to Figure 20-35.

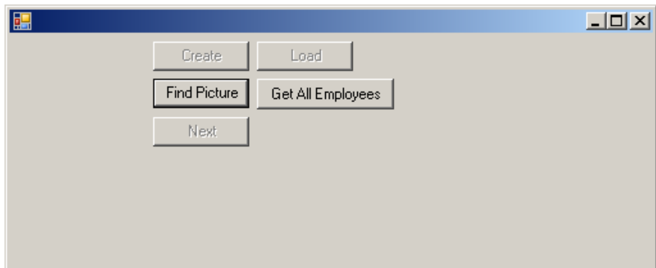


Figure 20-35: Initial State of the EmployeeTrainingServer Application Window

Referring to Figure 20-35 — you can trace the execution of the code as I discuss the use of this application. Initially, two buttons are enabled: Find Picture, and GetAllEmployees. Clicking the GetAllEmployees button calls the GetAllEmployees() event handler method, which creates an EmployeeDAO object and calls its GetAllEmployees() method. The `foreach` loop on line 121 then loops through the returned list of EmployeeVO objects and prints their information to the console. Clicking the Get All Employees button also enables the Next button, which is used to step through the EmployeeVO list (_list) by calling the NextEmployee() event handler method and to display employee pictures in the picture box. Note that with an initial load of test data there will be no employee pictures, so the test application code must properly handle the possibility of the EmployeeVO.Picture property being null.

Figure 20-36 shows an employee picture loaded, and the Create button enabled.

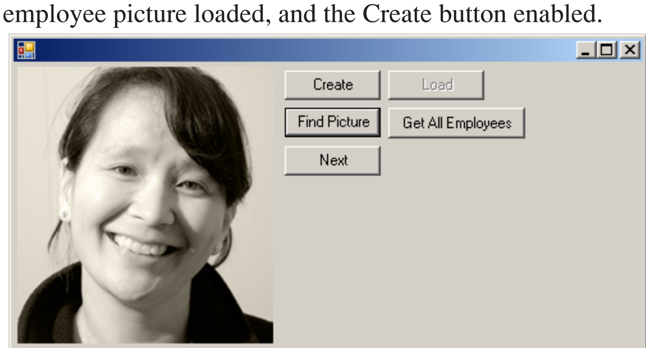


Figure 20-36: Employee Picture Loaded and Create Button Enabled

Referring to Figure 20-36 — to create a new employee and insert the picture into the database, click the Create button. To test the retrieval of an employee's data and picture click the Get All Employees button and then click the Next button until the picture appears in the PictureBox. Figure 20-37 shows several more employee pictures after they've been inserted and retrieved from the database.

Now, the employee ID photos I've been using are fairly small. It would be a good idea to try to load and retrieve a large image into the database. Figure 20-38 shows the results of that test.

This completes the development and testing phase of the first iteration. When you feel confident that the EmpoyeeDAO's insert and retrieval method's work fine you can move to the second iteration.

Second Iteration

A good set of objectives for the second iteration of the Employee Training application would be to finish the EmployeeDAO class by adding update and delete methods. You can also create a business object — a good name for

Figure 20-37: Testing with More Employee Pictures

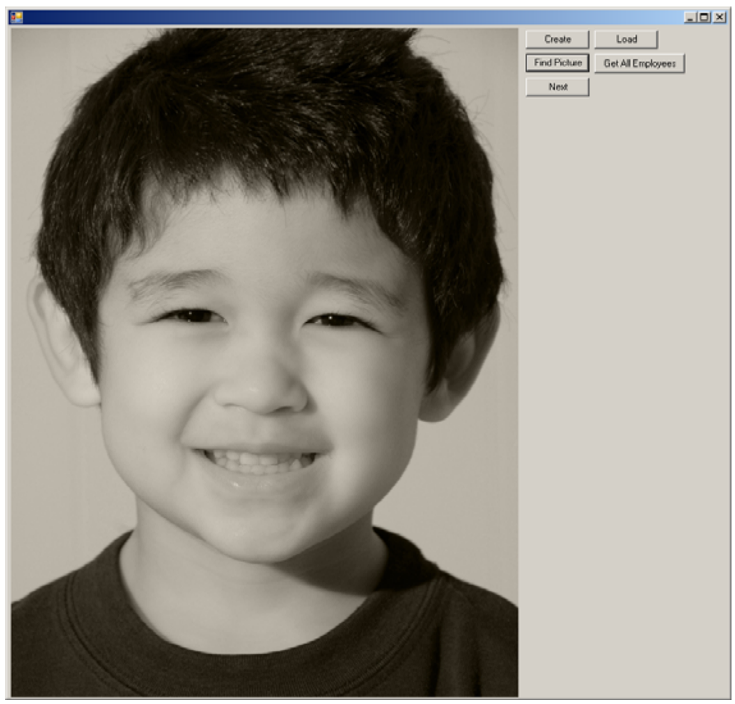


Figure 20-38: Testing the Insertion and Retrieval of a Large Image

which might be EmployeeAdminBO, and while you're at it create the TrainingDAO and TrainingVO classes. You might also want to add a few tweaks to the test application. Table 20-5 lists the design considerations and design decisions for this iteration.

Check-Off	Design Consideration	Design Decision
	DAO layer	Finish coding the EmployeeDAO. Add update and delete methods. Create the TrainingDAO class.
	Value objects	Create the TrainingVO class.
	BO layer	Create the EmployeeAdminBO class.

Table 20-5: Employee Training Server Application — Second Iteration Design Considerations And Decisions

Check-Off	Design Consideration	Design Decision
	Test application	Add the ability to add, update, and delete employee and employee training data. Modify the code to use the services of the EmployeeAdminBO class.
	Project file	Modify the EmployeeTrainingServer.proj file to build the contents of the bo directory.

Table 20-5: Employee Training Server Application — Second Iteration Design Considerations And Decisions

Figure 20-39 shows the UML class diagram for the TrainingDAO and TrainingVO classes.

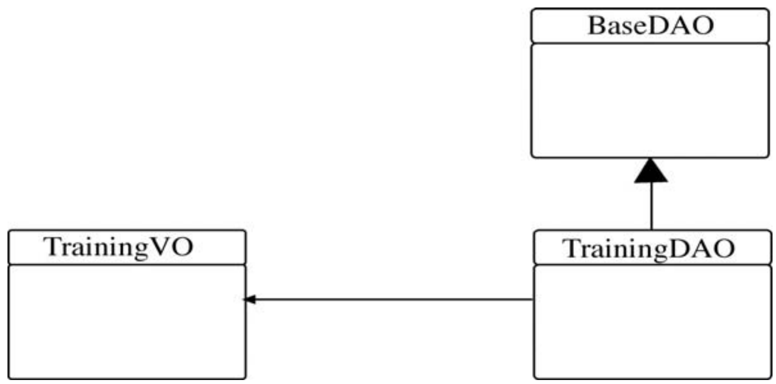


Figure 20-39: TrainingDAO and TrainingVO Class Diagram

Referring to Figure 20-39 — since the TrainingDAO class depends on the TrainingVO class, the TrainingVO class must be coded up first.

Figure 20-40 shows the UML diagram for the EmployeeAdminBO class.

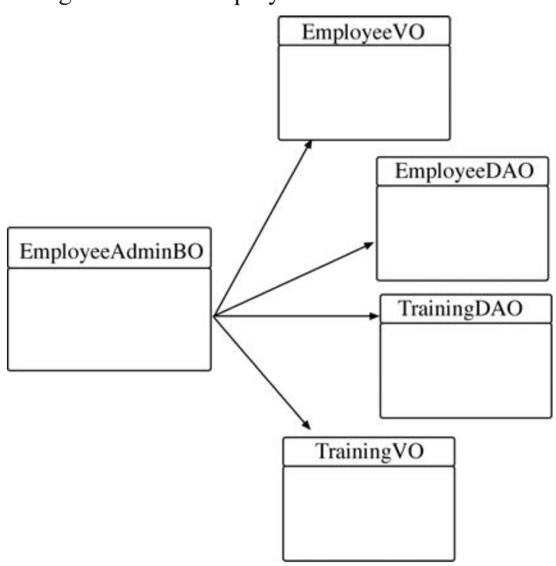


Figure 20-40: EmployeeAdminBO UML Class Diagram

Referring to Figure 20-40 — the EmployeeAdminBO class has dependencies on the EmployeeVO, EmployeeDAO, TrainingVO, and TrainingDAO classes. It would be a good idea to finish coding up these four classes before starting on the EmployeeAdminBO class.

Example 20.16 gives the code for the TrainingVO class.

20.16 TrainingVO.cs

```
1   using System;
2
3   namespace EmployeeTraining.VO {
4    [Serializable]
5     public class TrainingVO {
6
7       // Status enumeration
8       public enum TrainingStatus { Passed, Failed };
9       // Private fields
10      private int _trainingID;
11      private Guid _employeeID;
12      private String _title;
13      private String _description;
14      private DateTime _startdate;
15      private DateTime _enddate;
16      private TrainingStatus _status;
17
18      //Constructors
19      public TrainingVO(){}
20
21      public TrainingVO(int trainingID, Guid employeeID, String title, String description,
22                        DateTime startdate, DateTime enddate, TrainingStatus status){
23        TrainingID = trainingID;
24        EmployeeID = employeeID;
25        Title = title;
26        Description = description;
27        StartDate = startdate;
28        EndDate = enddate;
29        Status = status;
30      }
31
32      //Properties
33      public int TrainingID {
34        get { return _trainingID; }
35        set { _trainingID = value; }
36      }
37
38      public Guid EmployeeID {
39        get { return _employeeID; }
40        set { _employeeID = value; }
41      }
42
43      public String Title {
44        get { return _title; }
45        set { _title = value; }
46      }
47
48      public String Description {
49        get { return _description; }
50        set { _description = value; }
51      }
52
53      public DateTime StartDate {
54        get { return _startdate; }
55        set {_startdate = value; }
56      }
57
58      public DateTime EndDate {
59        get { return _enddate; }
60        set { _enddate = value; }
61      }
62
63      public TrainingStatus Status {
64        get { return _status; }
65        set { _status = value; }
66      }
67
68      public override String ToString(){
69        return Title + " " + Description + " " + EndDate.ToString() + " " + StartDate.ToString() +
70               " " + Status;
71      }
72
73    } // end class definition
74  } // end namespace
```

Referring to Example 20.16 — The TrainingVO class if fairly straightforward. I've added an enumeration named TrainingStatus on line 8 that contains two possible values: Passed and Failed. The TrainingStatus enumeration is used as the type for the Status property, which is defined on line 63.

Example 20-17 gives the code for the TrainingDAO class.

20.17 TrainingDAO.cs

```
using System;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Data.Sql;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using System.Collections.Generic;
using EmployeeTraining.VO;

using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Microsoft.Practices.EnterpriseLibrary.Data.Sql;

namespace EmployeeTraining.DAO {
  public class TrainingDAO : BaseDAO {
    //List of column identifiers used in perpared statements
    private const String TRAINING_ID = "@training_id";
    private const String EMPLOYEE_ID = "@employee_id";
    private const String TITLE = "@title";
    private const String DESCRIPTION = "@description";
    private const String STARTDATE = "@startdate";
    private const String ENDDATE = "@enddate";
    private const String STATUS = "@status";

    // SQL statement string constants
    private const String SELECT_ALL_COLUMNS =
      "SELECT trainingid, employeeid, title, description, startdate, enddate, status ";

    private const String SELECT_ALL_TRAINING =
      SELECT_ALL_COLUMNS +
      "FROM tbl_employee_training ";

    private const String SELECT_TRAINING_BY_TRAINING_ID =
      SELECT_ALL_TRAINING +
      "WHERE TrainingID = " + TRAINING_ID;

    private const String SELECT_TRAINING_BY_EMPLOYEE_ID =
      SELECT_ALL_TRAINING +
      "WHERE employeeid = " + EMPLOYEE_ID;

    private const String INSERT_TRAINING =
      "INSERT INTO tbl_employee_training " +
          "(EmployeeID, Title, Description, StartDate, EndDate, Status) " +
       "VALUES (" + EMPLOYEE_ID + ", " + TITLE + ", " + DESCRIPTION + ", " +
                    STARTDATE + ", " + ENDDATE + ", " + STATUS + ") " +
       "SELECT scope_identity()";

    private const String UPDATE_TRAINING =
      "UPDATE tbl_employee_training " +
      "SET EmployeeID = " + EMPLOYEE_ID + ", Title = " + TITLE + ", Description = " + DESCRIPTION +
             ", StartDate = " + STARTDATE + ", EndDate = " + ENDDATE + ", Status = " + STATUS + " " +
      "Where TrainingID = " + TRAINING_ID;

    private const String DELETE_TRAINING =
      "DELETE FROM tbl_employee_training " +
      "WHERE TrainingID = " + TRAINING_ID;

    private const String DELETE_TRAINING_FOR_EMPLOYEEID =
      "DELETE FROM tbl_employee_training " +
      "WHERE EmployeeID = " + EMPLOYEE_ID;

    // Public methods
 /***************************************************************************
   Gets a list of all training in the database.
 ****************************************************************************/
    public List<TrainingVO> GetAllTraining(){
      DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_TRAINING);
      return this.GetTrainingList(command);
    }

 /***********************************************
   Returns a TrainingVO object given a valid trainingid
 ************************************************/
    public TrainingVO GetTraining(int trainingid){
      DbCommand command = null;
      try{
        command = DataBase.GetSqlStringCommand(SELECT_TRAINING_BY_TRAINING_ID);
        DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingid);
```



```
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetTraining(command);
    }

 /***********************************************
    Returns a List<TrainingVO> object given a valid employeeid
 ***********************************************/
    public List<TrainingVO> GetTrainingForEmployee(Guid employeeid){
      DbCommand command = null;
      try{
        command = DataBase.GetSqlStringCommand(SELECT_TRAINING_BY_EMPLOYEE_ID);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetTrainingList(command);
    }

 /******************************************************************
    Inserts a row into tbl_employee_training given populated TrainingVO object.
    Returns fully-populated TrainingVO object, including primary key.
 ******************************************************************/
    public TrainingVO InsertTraining(TrainingVO trainingVO){
      int trainingID = 0;
      try{
       DbCommand command = DataBase.GetSqlStringCommand(INSERT_TRAINING);
       DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, trainingVO.EmployeeID);
       DataBase.AddInParameter(command, TITLE, DbType.String, trainingVO.Title);
       DataBase.AddInParameter(command, DESCRIPTION, DbType.String, trainingVO.Description);
       DataBase.AddInParameter(command, STARTDATE, DbType.DateTime, trainingVO.StartDate);
       DataBase.AddInParameter(command, ENDDATE, DbType.DateTime, trainingVO.EndDate);
       switch(trainingVO.Status){
         case TrainingVO.TrainingStatus.Passed :
              DataBase.AddInParameter(command, STATUS, DbType.String, "Passed");
              break;
         case TrainingVO.TrainingStatus.Failed :
              DataBase.AddInParameter(command, STATUS, DbType.String, "Failed");
              break;
       }
       trainingID = Convert.ToInt32(DataBase.ExecuteScalar(command));
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetTraining(trainingID);
    }

 /**************************************************************************
    Updates a row in the tbl_employee_training table given a populated TrainingVO object.
 **************************************************************************/
    public TrainingVO UpdateTraining(TrainingVO trainingVO){
      try{
       DbCommand command = DataBase.GetSqlStringCommand(UPDATE_TRAINING);
       DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingVO.TrainingID);
       DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, trainingVO.EmployeeID);
       DataBase.AddInParameter(command, TITLE, DbType.String, trainingVO.Title);
       DataBase.AddInParameter(command, DESCRIPTION, DbType.String, trainingVO.Description);
       DataBase.AddInParameter(command, STARTDATE, DbType.DateTime, trainingVO.StartDate);
       DataBase.AddInParameter(command, ENDDATE, DbType.DateTime, trainingVO.EndDate);
       switch(trainingVO.Status){
         case TrainingVO.TrainingStatus.Passed :
              DataBase.AddInParameter(command, STATUS, DbType.String, "Passed");
              break;
         case TrainingVO.TrainingStatus.Failed :
              DataBase.AddInParameter(command, STATUS, DbType.String, "Failed");
              break;
       }
       DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetTraining(trainingVO.TrainingID);
    }

 /**************************************************************************
    Deletes a row from the tbl_employee_training table for the given a training id.
 **************************************************************************/
    public void DeleteTraining(int trainingid){
      try {
        DbCommand command = DataBase.GetSqlStringCommand(DELETE_TRAINING);
```



```
        DataBase.AddInParameter(command, TRAINING_ID, DbType.Int32, trainingid);
        DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
    }

/*************************************************************************
   Deletes all training associated with given employee id.
**************************************************************************/
    public void DeleteTrainingForEmployeeID(Guid employeeid){
      try {
        DbCommand command = DataBase.GetSqlStringCommand(DELETE_TRAINING_FOR_EMPLOYEEID);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
        DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
    }

/*************************************************
   Private utility method that executes the given DbCommand
   and returns a fully-populated TrainingVO object
**************************************************/
    private TrainingVO GetTraining(DbCommand command){
      TrainingVO trainingVO = null;
      IDataReader reader = null;
      try {
        reader = DataBase.ExecuteReader(command);
        if(reader.Read()){
          trainingVO = this.FillInTrainingVO(reader);
        }
      } catch(Exception e){
        Console.WriteLine(e);
      } finally {
        base.CloseReader(reader);
      }
      return trainingVO;
    }

/***********************************************************
   Private utility method that gets a list of TrainingVOs given a DbCommand
*****************************************************************/
    private List<TrainingVO> GetTrainingList(DbCommand command){
      IDataReader reader = null;
      List<TrainingVO> training_list = new List<TrainingVO>();
      try{
        reader = DataBase.ExecuteReader(command);
        while(reader.Read()){
          TrainingVO trainingVO = this.FillInTrainingVO(reader);
          training_list.Add(trainingVO);
        }
      } catch(Exception e){
        Console.WriteLine(e);
      } finally{
        base.CloseReader(reader);
      }
      return training_list;
    }

/*******************************************************************
   Private utility method that fills in a TrainingVO
********************************************************************/
    private TrainingVO FillInTrainingVO(IDataReader reader){
      TrainingVO trainingVO = new TrainingVO();
      trainingVO.TrainingID = reader.GetInt32(0);
      trainingVO.EmployeeID = reader.GetGuid(1);
      trainingVO.Title = reader.GetString(2);
      trainingVO.Description = reader.GetString(3);
      trainingVO.StartDate = reader.GetDateTime(4);
      trainingVO.EndDate = reader.GetDateTime(5);
      String status = reader.GetString(6);
      switch(status){
       case "Passed" : trainingVO.Status = TrainingVO.TrainingStatus.Passed;
                       break;
       case "Failed" : trainingVO.Status = TrainingVO.TrainingStatus.Failed;
                       break;
      }
      return trainingVO;
    }

```



```
  } // end class definition
} // end namespace
```

Referring to Example 20.17 — the TrainingDAO class inserts, queries, updates, and deletes data in the tbl_employee_training table. This class functions like the EmployeeDAO so I'll let you walk through the code on your own.

Example 20.18 gives the completed version of the EmployeeDAO class with the delete and update methods added.

20.18 EmployeeDAO.cs (Complete)

```
using System;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Data.Sql;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using EmployeeTraining.VO;

using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Microsoft.Practices.EnterpriseLibrary.Data.Sql;

namespace EmployeeTraining.DAO {
  public class EmployeeDAO : BaseDAO {

   private bool debug = true;

    //List of column identifiers used in perpared statements
    private const String EMPLOYEE_ID = "@employee_id";
    private const String FIRST_NAME = "@first_name";
    private const String MIDDLE_NAME = "@middle_name";
    private const String LAST_NAME = "@last_name";
    private const String BIRTHDAY = "@birthday";
    private const String GENDER = "@gender";
    private const String PICTURE = "@picture";

    private const String SELECT_ALL_COLUMNS =
      "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";

    private const String SELECT_ALL_EMPLOYEES =
      SELECT_ALL_COLUMNS +
      "FROM tbl_employee ";

    private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
      SELECT_ALL_EMPLOYEES +
      "WHERE employeeid = " + EMPLOYEE_ID;


    private const String INSERT_EMPLOYEE =
      "INSERT INTO tbl_employee " +
          "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
       "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
                    BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";

    private const String UPDATE_EMPLOYEE =
      "UPDATE tbl_employee " +
      "SET FirstName = " + FIRST_NAME + ", MiddleName = " + MIDDLE_NAME + ", LastName = " + LAST_NAME +
             ", Birthday = " + BIRTHDAY + ", Gender = " + GENDER + ", Picture = " + PICTURE + " " +
      "WHERE EmployeeID = " + EMPLOYEE_ID;

    private const String DELETE_EMPLOYEE =
      "DELETE FROM tbl_employee " +
      "WHERE EmployeeID = " + EMPLOYEE_ID;

 /************************************
   Returns a List<EmployeeVO> object
 **************************************/
    public List<EmployeeVO> GetAllEmployees(){
      DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
      return this.GetEmployeeList(command);
    }

 /***********************************************
   Returns an EmployeeVO object given a valid employeeid
 *************************************************/
    public EmployeeVO GetEmployee(Guid employeeid){
```

```
      DbCommand command = null;
      try{
        command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetEmployee(command);
    }

  /*******************************************************
    Inserts an employee given a fully-populated EmployeeVO object
  ********************************************************/
    public EmployeeVO InsertEmployee(EmployeeVO employee){
      try{
        employee.EmployeeID = Guid.NewGuid();
        DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
        DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
        DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
        DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
        DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
        switch(employee.Gender){
          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
               break;
          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
               break;
        }

        if(employee.Picture != null){
         if(debug){ Console.WriteLine("Inserting picture!"); }
          MemoryStream ms = new MemoryStream();
          employee.Picture.Save(ms, ImageFormat.Tiff);
          byte[] byte_array = ms.ToArray();
          if(debug){
            for(int i=0; i<byte_array.Length; i++){
              Console.Write(byte_array[i]);
            }
          } // end if debug
         DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
         if(debug){ Console.WriteLine("Picture inserted, I think!"); }
        }

        DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetEmployee(employee.EmployeeID);
    }

  ********************************************************
    Updates a row in the tbl_employee table given the fully-populated
    EmployeeVO object.
  *********************************************************/
    public EmployeeVO UpdateEmployee(EmployeeVO employee){
      try {
        DbCommand command = DataBase.GetSqlStringCommand(UPDATE_EMPLOYEE);
        DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
        DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
        DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
        DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
        switch(employee.Gender){
          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
               break;
          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
               break;
        }
        if(employee.Picture != null){
         if(debug){ Console.WriteLine("Inserting picture!"); }
          MemoryStream ms = new MemoryStream();
          employee.Picture.Save(ms, ImageFormat.Tiff);
          byte[] byte_array = ms.ToArray();
          if(debug){
            for(int i=0; i<byte_array.Length; i++){
              Console.Write(byte_array[i]);
            }
          } // end if debug
         DataBase.AddInParameter(command, PICTURE, DbType.Binary, byte_array);
         if(debug){ Console.WriteLine("Picture inserted, I think!"); }
        }
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
```



```
       DataBase.ExecuteNonQuery(command);
   } catch(Exception e){
     Console.WriteLine(e);
   }
   return this.GetEmployee(employee.EmployeeID);
   }

/********************************************************
   Deletes a row from the tbl_employee table given an employee id.
**********************************************************/
  public void DeleteEmployee(Guid employeeid){
    try{
      DbCommand command = DataBase.GetSqlStringCommand(DELETE_EMPLOYEE);
      DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
      DataBase.ExecuteNonQuery(command);
    } catch(Exception e){
      Console.WriteLine(e);
    }
  }

/************************************************
   Private utility method that executes the given DbCommand
   and returns a fully-populated EmployeeVO object
**************************************************/
   private EmployeeVO GetEmployee(DbCommand command){
     EmployeeVO empVO = null;
     IDataReader reader = null;
     try {
       reader = DataBase.ExecuteReader(command);
       if(reader.Read()){
         empVO = this.FillInEmployeeVO(reader);
       }
     } catch(Exception e){
       Console.WriteLine(e);
     } finally {
       base.CloseReader(reader);
     }
     return empVO;
   }

/****************************************************
   GetEmployeeList() - returns a List<EmployeeVO> object
*******************************************************/
   private List<EmployeeVO> GetEmployeeList(DbCommand command){
     IDataReader reader = null;
     List<EmployeeVO> employee_list = new List<EmployeeVO>();
     try{
       reader = DataBase.ExecuteReader(command);
       while(reader.Read()){
         EmployeeVO empVO = this.FillInEmployeeVO(reader);
         employee_list.Add(empVO);
       }
     } catch(Exception e){
       Console.WriteLine(e);
     } finally{
       base.CloseReader(reader);
     }
     return employee_list;
   }

/********************************************************
   Private utility method that populates an EmployeeVO object from
   data read from the IDataReader object
*************************************************************/
   private EmployeeVO FillInEmployeeVO(IDataReader reader){
     EmployeeVO empVO = new EmployeeVO();
     empVO.EmployeeID = reader.GetGuid(0);
     empVO.FirstName = reader.GetString(1);
     empVO.MiddleName = reader.GetString(2);
     empVO.LastName = reader.GetString(3);
     empVO.BirthDay = reader.GetDateTime(4);
     String gender = reader.GetString(5);
     switch(gender){
       case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
                  break;
       case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
                  break;
     }
     if(!reader.IsDBNull(6)){
       int buffersize = 5000;
       int startindex = 0;
```



```
      Byte[] byte_array = new Byte[buffersize];
      MemoryStream ms = new MemoryStream();
      long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
      while(retval > 0){
        ms.Write(byte_array, 0, byte_array.Length);
        startindex += buffersize;
        retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
       }
      empVO.Picture = new Bitmap(ms);
    }
    return empVO;
  }

 } // end EmployeeDAO definition
} // end namespace
```

Example 20.19 gives the code for the EmployeeAdminBO class.

20.19 EmployeeAdminBO.cs

```
using System;
using System.Collections.Generic;
using EmployeeTraining.VO;
using EmployeeTraining.DAO;

namespace EmployeeTraining.BO {
  public class EmployeeAdminBO {

  #region Employee Methods

   public EmployeeVO CreateEmployee(EmployeeVO employee){
     EmployeeDAO dao = new EmployeeDAO();
     return dao.InsertEmployee(employee);
   }

   public EmployeeVO GetEmployee(Guid employeeID){
     EmployeeDAO dao = new EmployeeDAO();
     return dao.GetEmployee(employeeID);
   }

   public List<EmployeeVO> GetAllEmployees(){
     EmployeeDAO dao = new EmployeeDAO();
     return dao.GetAllEmployees();
   }

   public EmployeeVO UpdateEmployee(EmployeeVO employee){
     EmployeeDAO dao = new EmployeeDAO();
     return dao.UpdateEmployee(employee);
   }

   public void DeleteEmployee(Guid employeeID){
     EmployeeDAO dao = new EmployeeDAO();
     dao.DeleteEmployee(employeeID);
   }
   #endregion Employee Methods

   #region Training Methods
   public TrainingVO CreateTraining(TrainingVO training){
     TrainingDAO dao = new TrainingDAO();
     return dao.InsertTraining(training);
   }

   public TrainingVO GetTraining(int trainingID){
     TrainingDAO dao = new TrainingDAO();
     return dao.GetTraining(trainingID);
   }

   public List<TrainingVO> GetTrainingForEmployee(Guid employeeID){
     TrainingDAO dao = new TrainingDAO();
     return dao.GetTrainingForEmployee(employeeID);
   }

   public TrainingVO UpdateTraining(TrainingVO training){
     TrainingDAO dao = new TrainingDAO();
     return dao.UpdateTraining(training);
   }

   public void DeleteTrainingForEmployee(EmployeeVO employee){
     TrainingDAO dao = new TrainingDAO();
     dao.DeleteTrainingForEmployeeID(employee.EmployeeID);
   }

   public void DeleteTraining(int trainingID){
```



```
64        TrainingDAO dao = new TrainingDAO();
65        dao.DeleteTraining(trainingID);
66      }
67      #endregion Training Methods
68
69    } // End class definition
70  } // End namespace
```

Referring to Example 20.19 — the EmployeeAdminBO provides methods to create, query, update, and delete employee and employee training data. The methods that deal with employee data have been grouped in the Employee Methods region by using the `#region` and `#endregion` directives. Regions allow you to collapse and expand sections of code when using Visual Studio or a compatible text editor like Notepad++. Figure 20-41 shows how code regions look when they are collapsed in Notepad++.

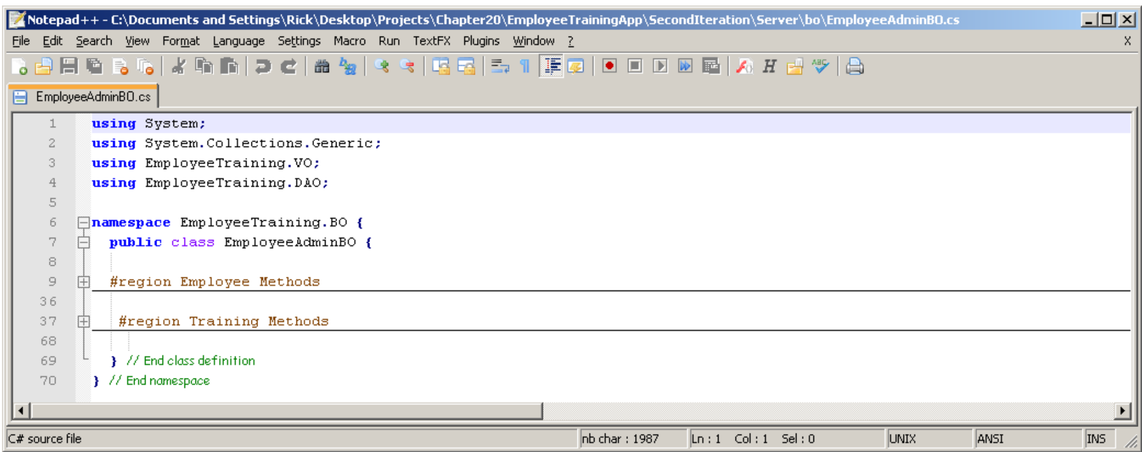


Figure 20-41: Collapsed Code Regions in Notepad++

Referring again to Example 20.19 — in this example, all the methods are short because they are simply pass-through methods to the appropriate DAO. For example, the CreateEmployee() method defined on line 11 creates an instance of the EmployeeDAO class and calls its InsertEmployee() method to insert the EmployeeVO object's data into the tbl_employee table. In a more real world example, the EmployeeAdminBo would be used to implement and enforce more elaborate business rules. For example, if only certain types of users were allowed to create, update, or delete employee data, then those corresponding methods would perform the requisite checks to validate the user's credentials before allowing the insert, update, or delete operations via the DAO to occur.

OK, before you can compile the EmployeeAdminBO class you must make a change to the MSBuild project file. Example 20.20 shows the updated EmployeeTrainingServer.proj file.

20.20 EmployeeTrainingServer.proj (Mod 1)

```
<Project DefaultTargets="CompileApp"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">


   <PropertyGroup>
     <IncludeDebugInformation>false</IncludeDebugInformation>
     <BuildDir>build</BuildDir>
     <LibDir>lib</LibDir>
     <AppDir>app</AppDir>
     <RefDir>ref</RefDir>
     <ConfigDir>config</ConfigDir>
   </PropertyGroup>

    <ItemGroup>
      <DAO Include="dao\**\*.cs" />
      <BO Include="bo\**\*.cs" />
      <VO Include="vo\**\*.cs" />
      <APP Include="app\**\*.cs" />
      <LIB Include="lib\**\*.dll" />
      <REF Include="ref\**\*.dll" />
      <CONFIG Include="config\**\*.config" />
      <EXE Include="app\**\*.exe" />
    </ItemGroup>

    <Target Name="MakeDirs">
      <MakeDir Directories="$(BuildDir)" />
      <MakeDir Directories="$(LibDir)" />
```



```
28      </Target>
29
30      <Target Name="RemoveDirs">
31        <RemoveDir Directories="$(BuildDir)" />
32        <RemoveDir Directories="$(LibDir)" />
33      </Target>
34
35      <Target Name="Clean"
36              DependsOnTargets="RemoveDirs;MakeDirs">
37      </Target>
38
39       <Target Name="CopyFiles">
40         <Copy
41           SourceFiles="@(CONFIG);@(LIB);@(REF)"
42           DestinationFolder="$(BuildDir)" />
43      </Target>
44
45      <Target Name="CompileVO"
46              Inputs="@(VO)"
47              Outputs="$(LibDir)\VOLib.dll">
48        <Csc Sources="@(VO)"
49            TargetType="library"
50            References="@(REF);@(LIB)"
51            OutputAssembly="$(LibDir)\VOLib.dll">
52        </Csc>
53     </Target>
54
55      <Target Name="CompileDAO"
56              Inputs="@(DAO)"
57              Outputs="$(LibDir)\DAOLib.dll"
58              DependsOnTargets="CompileVO">
59         <Csc Sources="@(DAO)"
60            TargetType="library"
61            References="@(REF);@(LIB)"
62            WarningLevel="0"
63            OutputAssembly="$(LibDir)\DAOLib.dll">
64        </Csc>
65      </Target>
66
67      <Target Name="CompileBO"
68              Inputs="@(BO)"
69              Outputs="$(LibDir)\BOLib.dll"
70              DependsOnTargets="CompileDAO">
71        <Csc Sources="@(BO)"
72            TargetType="library"
73            References="@(REF);@(LIB)"
74            WarningLevel="0"
75            OutputAssembly="$(LibDir)\BOLib.dll">
76        </Csc>
77      </Target>
78
79      <Target Name="CompileApp"
80              Inputs="@(APP)"
81              Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
82              DependsOnTargets="CompileBO">
83        <Csc Sources="@(APP)"
84             TargetType="exe"
85             References="@(REF);@(LIB)"
86             OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
87        </Csc>
88      </Target>
89
90      <Target Name="CompileAll">
91        <Csc Sources="@(VO);@(DAO);@(BO);@(APP)"
92             TargetType="exe"
93             References="@(REF);@(LIB)"
94             OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
95        </Csc>
96      </Target>
97
98      <Target Name="Run"
99              DependsOnTargets="CompileApp;CopyFiles">
100       <Exec Command="$(MSBuildProjectName).exe"
101             WorkingDirectory="$(BuildDir)" />
102     </Target>
103 </Project>
```

Referring to Example 20.20 — the only changes made to the file were to the DefaultTargets on line 1, which is now set to CompileApp, and to the DependOnTargets in the CompileApp target, which is now set to CompileBO.

Testing The Code - Second Iteration

To completely test the code developed thus far requires major enhancements to the test application. Figure 20-42 shows the modified user interface of the EmployeeTrainingServer applicaton.

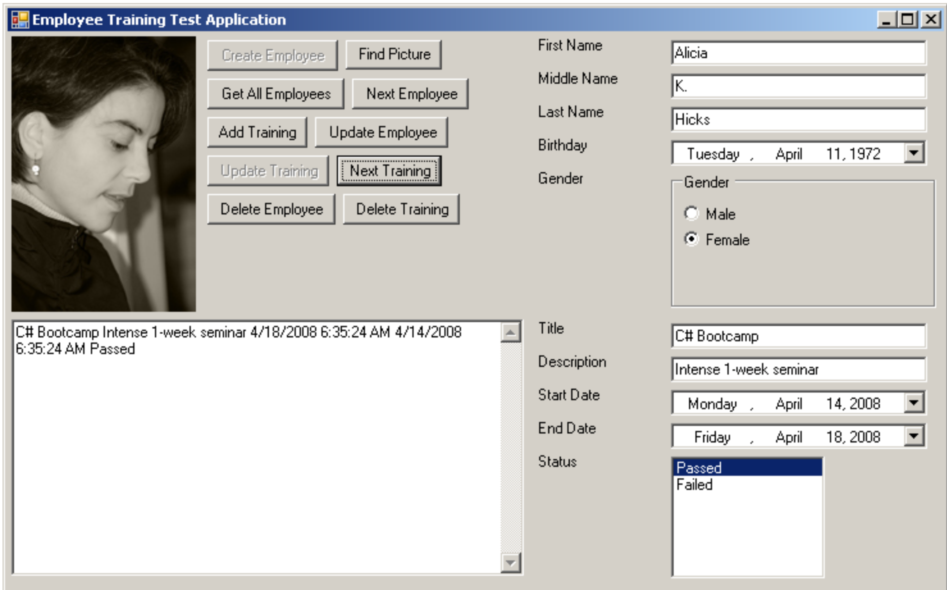


Figure 20-42: Modified Test Application

Referring to Figure 20-42 — the test application has been enhanced to allow the creation of employee training records as well as the ability to update and delete both employee and training data. The code for this version of the test application is given in Example 20.21.

20.21 EmployeeTrainingServer.cs (Test Application 2nd Iteration)

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Imaging;
using System.Collections.Generic;
using EmployeeTraining.BO;
using EmployeeTraining.VO;

public class EmployeeTrainingServer : Form  {

  private PictureBox _picturebox;
  private TableLayoutPanel _tablepanel;

  private FlowLayoutPanel _flowpanel;
  private Button _create_employee_botton;
  private Button _find_picture_button;
  private Button _get_all_employees_button;
  private Button _next_employee_button;
  private Button _add_training_button;
  private Button _update_employee_button;
  private Button _update_training_button;
  private Button _next_training_button;
  private Button _delete_employee_button;
  private Button _delete_training_button;

  private TableLayoutPanel _employee_info_entry_panel;
  private Label _fname_label;
  private Label _mname_label;
  private Label _lname_label;
  private Label _bday_label;
  private Label _gender_label;
  private TextBox _fname_textbox;
  private TextBox _mname_textbox;
  private TextBox _lname_textbox;
  private DateTimePicker _bday_picker;
  private GroupBox _gender_groupbox;
  private RadioButton _male_button;
```

```
    private RadioButton _female_button;


    private const int TABLE_PANEL_ROW_COUNT = 2;
    private const int TABLE_PANEL_COLUMN_COUNT = 3;
    private const int TABLE_PANEL_HEIGHT = 600;
    private const int TABLE_PANEL_WIDTH = 600;
    private const int EMPLOYEE_INFO_PANEL_HEIGHT = 200;
    private const int EMPLOYEE_INFO_PANEL_WIDTH= 200;
    private const int EMPLOYEE_INFO_PANEL_ROW_COUNT = 5;
    private const int EMPLOYEE_INFO_PANEL_COLUMN_COUNT = 2;
    private const int TRAINING_INFO_PANEL_ROW_COUNT = 5;
    private const int TRAINING_INFO_PANEL_COLUMN_COUNT = 2;
    private const int TRAINING_INFO_PANEL_HEIGHT = 200;
    private const int TRAINING_INFO_PANEL_WIDTH = 200;
    private const int TEXTBOX_WIDTH = 200;
    private const int SMALL_PADDING = 100;
    private const int LARGE_PADDING = 150;
    private const int TRAINING_TEXTBOX_WIDTH = 400;
    private const int TRAINING_TEXTBOX_HEIGHT = 200;
    private const int PICTUREBOX_WIDTH = 150;
    private const int PICTUREBOX_HEIGHT = 150;
    private const int GROUPBOX_WIDTH = 200;
    private const int GROUPBOX_HEIGHT = 125;

    private TableLayoutPanel _training_info_entry_panel;
    private Label _title_label;
    private Label _description_label;
    private Label _startdate_label;
    private Label _enddate_label;
    private Label _status_label;
    private TextBox _title_textbox;
    private TextBox _description_textbox;
    private DateTimePicker _startdate_picker;
    private DateTimePicker _enddate_picker;
    private ListBox _status_listbox;

    private TextBox _training_textbox;

    private EmployeeVO _emp_vo;
    private List<EmployeeVO> _employee_list;
    private List<TrainingVO> _training_list;
    private int _next_employee = 0;
    private int _next_training = 0;
    private OpenFileDialog _dialog;

    public EmployeeTrainingServer(){
      this.InitializeComponent();
      Application.Run(this);
    }

    private void InitializeComponent(){
       this.SuspendLayout();
      _tablepanel = new TableLayoutPanel();
      _flowpanel = new FlowLayoutPanel();
      _flowpanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
      _tablepanel.SuspendLayout();
      _tablepanel.RowCount = TABLE_PANEL_ROW_COUNT;
      _tablepanel.ColumnCount = TABLE_PANEL_COLUMN_COUNT;
      _tablepanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
      _tablepanel.Dock = DockStyle.Top;

      _picturebox = new PictureBox();
      _picturebox.Height = PICTUREBOX_WIDTH;
      _picturebox.Width = PICTUREBOX_HEIGHT;
      _picturebox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;

      _create_employee_botton = new Button();
      _create_employee_botton.Text = "Create Employee";
      _create_employee_botton.AutoSize = true;
      _create_employee_botton.Click += this.CreateEmployee;
      _create_employee_botton.Enabled = false;

      _find_picture_button = new Button();
      _find_picture_button.Text = "Find Picture";
      _find_picture_button.Click += this.ShowOpenFileDialog;

      _get_all_employees_button = new Button();
      _get_all_employees_button.Text = "Get All Employees";
      _get_all_employees_button.AutoSize = true;
      _get_all_employees_button.Click += this.GetAllEmployees;
```



```
      _next_employee_button = new Button();
      _next_employee_button.Text = "Next Employee";
      _next_employee_button.AutoSize = true;
      _next_employee_button.Click += this.NextEmployee;
      _next_employee_button.Enabled = false;

      _add_training_button = new Button();
      _add_training_button.Text = "Add Training";
      _add_training_button.AutoSize = true;
      _add_training_button.Click += this.AddTraining;
      _add_training_button.Enabled = false;

      _update_employee_button = new Button();
      _update_employee_button.Text = "Update Employee";
      _update_employee_button.AutoSize = true;
      _update_employee_button.Click += this.UpdateEmployee;
      _update_employee_button.Enabled = false;

      _update_training_button = new Button();
      _update_training_button.Text = "Update Training";
      _update_training_button.AutoSize = true;
      _update_training_button.Click += this.UpdateTraining;
      _update_training_button.Enabled = false;

      _next_training_button = new Button();
      _next_training_button.Text = "Next Training";
      _next_training_button.AutoSize = true;
      _next_training_button.Click += this.NextTraining;
      _next_training_button.Enabled = false;

      _delete_employee_button = new Button();
      _delete_employee_button.Text = "Delete Employee";
      _delete_employee_button.AutoSize = true;
      _delete_employee_button.Click += this.DeleteEmployee;
      _delete_employee_button.Enabled = false;

      _delete_training_button = new Button();
      _delete_training_button.Text = "Delete Training";
      _delete_training_button.AutoSize = true;
      _delete_training_button.Click += this.DeleteTraining;
      _delete_training_button.Enabled = false;

      _tablepanel.Controls.Add(_picturebox);
      _flowpanel.Controls.Add(_create_employee_botton);
      _flowpanel.Controls.Add(_find_picture_button);
      _flowpanel.Controls.Add(_get_all_employees_button);
      _flowpanel.Controls.Add(_next_employee_button);
      _flowpanel.Controls.Add(_add_training_button);
      _flowpanel.Controls.Add(_update_employee_button);
      _flowpanel.Controls.Add(_update_training_button);
      _flowpanel.Controls.Add(_next_training_button);
      _flowpanel.Controls.Add(_delete_employee_button);
      _flowpanel.Controls.Add(_delete_training_button);

      _tablepanel.Controls.Add(_flowpanel);

      _employee_info_entry_panel = new TableLayoutPanel();
      _employee_info_entry_panel.SuspendLayout();
      _employee_info_entry_panel.Height = EMPLOYEE_INFO_PANEL_HEIGHT;
      _employee_info_entry_panel.Width = EMPLOYEE_INFO_PANEL_WIDTH;
      _employee_info_entry_panel.RowCount = EMPLOYEE_INFO_PANEL_ROW_COUNT;
      _employee_info_entry_panel.ColumnCount = EMPLOYEE_INFO_PANEL_COLUMN_COUNT;
      _fname_label = new Label();
      _fname_label.Text = "First Name";
      _mname_label = new Label();
      _mname_label.Text = "Middle Name";
      _lname_label = new Label();
      _lname_label.Text = "Last Name";
      _bday_label = new Label();
      _bday_label.Text = "Birthday";
      _gender_label = new Label();
      _gender_label.Text = "Gender";
      _fname_textbox = new TextBox();
      _fname_textbox.Width = TEXTBOX_WIDTH;
      _mname_textbox = new TextBox();
      _mname_textbox.Width = TEXTBOX_WIDTH;
      _lname_textbox = new TextBox();
      _lname_textbox.Width = TEXTBOX_WIDTH;
      _bday_picker = new DateTimePicker();
      _gender_groupbox = new GroupBox();
```



```
     _gender_groupbox.Text = "Gender";
     _gender_groupbox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
                    | AnchorStyles.Right;
     _gender_groupbox.Height = GROUPBOX_HEIGHT;
     _gender_groupbox.Width = GROUPBOX_WIDTH;

     _male_button = new RadioButton();
     _male_button.Text = "Male";
     _male_button.Checked = true;
     _male_button.Location = new Point(10, 20);
     _female_button = new RadioButton();
     _female_button.Text = "Female";
     _female_button.Location = new Point(10, 40);
     _gender_groupbox.Controls.Add(_male_button);
     _gender_groupbox.Controls.Add(_female_button);
     _gender_groupbox.Size = new Size(50, 50);
     _employee_info_entry_panel.Controls.Add(_fname_label);
     _employee_info_entry_panel.Controls.Add(_fname_textbox);
     _employee_info_entry_panel.Controls.Add(_mname_label);
     _employee_info_entry_panel.Controls.Add(_mname_textbox);
     _employee_info_entry_panel.Controls.Add(_lname_label);
     _employee_info_entry_panel.Controls.Add(_lname_textbox);
     _employee_info_entry_panel.Controls.Add(_bday_label);
     _employee_info_entry_panel.Controls.Add(_bday_picker);
     _employee_info_entry_panel.Controls.Add(_gender_label);
     _employee_info_entry_panel.Controls.Add(_gender_groupbox);
     _employee_info_entry_panel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
                           | AnchorStyles.Right;

     _tablepanel.Controls.Add(_employee_info_entry_panel);

    _training_info_entry_panel = new TableLayoutPanel();
    _training_info_entry_panel.RowCount = TRAINING_INFO_PANEL_ROW_COUNT;
    _training_info_entry_panel.ColumnCount = TRAINING_INFO_PANEL_COLUMN_COUNT;
    _training_info_entry_panel.Height = TRAINING_INFO_PANEL_HEIGHT;
    _training_info_entry_panel.Width = TRAINING_INFO_PANEL_WIDTH;
    _title_label = new Label();
    _title_label.Text = "Title";
    _description_label = new Label();
    _description_label.Text = "Description";
    _startdate_label = new Label();
    _startdate_label.Text = "Start Date";
    _enddate_label = new Label();
    _enddate_label.Text = "End Date";
    _status_label = new Label();
    _status_label.Text = "Status";
    _title_textbox = new TextBox();
    _title_textbox.Width = TEXTBOX_WIDTH;
    _description_textbox = new TextBox();
    _description_textbox.Width = TEXTBOX_WIDTH;
    _startdate_picker = new DateTimePicker();
    _enddate_picker = new DateTimePicker();
    _status_listbox = new ListBox();
    _status_listbox.Items.Add("Passed");
    _status_listbox.Items.Add("Failed");
    _status_listbox.SetSelected(0, true);

    _training_info_entry_panel.Controls.Add(_title_label);
    _training_info_entry_panel.Controls.Add(_title_textbox);
    _training_info_entry_panel.Controls.Add(_description_label);
    _training_info_entry_panel.Controls.Add(_description_textbox);
    _training_info_entry_panel.Controls.Add(_startdate_label);
    _training_info_entry_panel.Controls.Add(_startdate_picker);
    _training_info_entry_panel.Controls.Add(_enddate_label);
    _training_info_entry_panel.Controls.Add(_enddate_picker);
    _training_info_entry_panel.Controls.Add(_status_label);
    _training_info_entry_panel.Controls.Add(_status_listbox);
    _training_info_entry_panel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
                         | AnchorStyles.Right;

    _tablepanel.Controls.Add(_training_info_entry_panel);

    _training_textbox = new TextBox();
    _training_textbox.Multiline = true;
    _training_textbox.ScrollBars = ScrollBars.Vertical;
    _training_textbox.Dock = DockStyle.Top;
    _training_textbox.Width = TRAINING_TEXTBOX_WIDTH;
    _training_textbox.Height = TRAINING_TEXTBOX_HEIGHT;
    _tablepanel.Controls.Add(_training_textbox);
    _tablepanel.SetRow(_training_textbox, 1);
    _tablepanel.SetColumn(_training_textbox, 0);
```



```
    _tablepanel.SetColumnSpan(_training_textbox, 2);

    this.Controls.Add(_tablepanel);
    _tablepanel.Width =  _training_textbox.Width + _employee_info_entry_panel.Width + LARGE_PADDING;
    _tablepanel.Height = TABLE_PANEL_HEIGHT;
    this.Width = _tablepanel.Width;
    this.Height = _tablepanel.Height;
    this.Text = "Employee Training Test Application";
    _employee_info_entry_panel.ResumeLayout();
    _tablepanel.ResumeLayout();
    this.ResumeLayout();
    _dialog = new OpenFileDialog();
    _dialog.FileOk += this.LoadPicture;
   }

   public void ShowOpenFileDialog(Object sender, EventArgs e){
     this.ResetEntryFields();
     this.ResetTrainingTextbox();
     _add_training_button.Enabled = false;
     _delete_employee_button.Enabled = false;
     _update_employee_button.Enabled = false;
     _next_training_button.Enabled = false;
     _dialog.ShowDialog();
   }

   public void LoadPicture(Object sender, EventArgs e){
      String filename = _dialog.FileName;
     _picturebox.Image = new Bitmap(filename);
     this.AdjustAppWindowSize();
     _create_employee_botton.Enabled = true;
   }

   public void CreateEmployee(Object sender, EventArgs e){
     EmployeeVO vo = new EmployeeVO();
     vo = this.PopulateEmployeeVOFromEntryFields(vo);

     EmployeeAdminBO bo = new EmployeeAdminBO();
     _emp_vo = bo.CreateEmployee(vo);
     _picturebox.Image = null;
     _create_employee_botton.Enabled = false;
     this.ResetEntryFields();
     this.DisplayEmployeeInfo();
     this.DisplayEmployeeTraining(bo);
   }

   public void GetAllEmployees(Object sender, EventArgs e){
     EmployeeAdminBO bo = new EmployeeAdminBO();
     _employee_list = bo.GetAllEmployees();
     foreach(EmployeeVO emp in _employee_list){
       Console.WriteLine(emp);
     }
     _next_employee_button.Enabled = true;
   }

   public void NextEmployee(Object sender, EventArgs e){
     _next_employee++;
     _next_training = 0;
     Console.WriteLine(_next_employee);
     if(_next_employee >= _employee_list.Count){
       _next_employee = 0;
     }
     Console.WriteLine(_next_employee);
     if(_employee_list.Count > 0){
       Console.WriteLine(_employee_list[_next_employee]);
       _emp_vo = _employee_list[_next_employee];
       this.DisplayEmployeeInfo();
       this.DisplayEmployeeTraining(new EmployeeAdminBO());
       if(_training_list.Count > 0){
         _update_training_button.Enabled = true;
         _next_training_button.Enabled = true;
       }else{
         _update_training_button.Enabled = false;
         _next_training_button.Enabled = false;
         _delete_training_button.Enabled = false;
       }
       _delete_employee_button.Enabled = true;
       _add_training_button.Enabled = true;
       _update_employee_button.Enabled = true;
     }else{
       _delete_employee_button.Enabled = false;
       _add_training_button.Enabled = false;
```



```
      _update_employee_button.Enabled = false;
    }
    this.ResetTrainingEntryFields();
  }

  public void UpdateEmployee(Object sender, EventArgs e){
    _emp_vo = this.PopulateEmployeeVOFromEntryFields(_emp_vo);
    EmployeeAdminBO bo = new EmployeeAdminBO();
    _emp_vo = bo.UpdateEmployee(_emp_vo);
    this.ResetEntryFields();
    this.DisplayEmployeeInfo();
    this.DisplayEmployeeTraining(bo);
  }

  public void AddTraining(Object sender, EventArgs e){
    TrainingVO vo = new TrainingVO();
    vo = this.PopulateTrainingVOFromEntryFields(vo);
    EmployeeAdminBO bo = new EmployeeAdminBO();
    bo.CreateTraining(vo);
    this.DisplayEmployeeTraining(bo);
    this.ResetTrainingEntryFields();
    _next_training_button.Enabled = true;
  }

  public void NextTraining(Object Sender, EventArgs e){
    _next_training++;
    if(_next_training >= _training_list.Count){
      _next_training = 0;
    }
    if(_training_list.Count > 0){
      this.DisplayTrainingInfo(_training_list[_next_training]);
      _delete_training_button.Enabled = true;
    }
  }

  public void UpdateTraining(Object Sender, EventArgs e){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    bo.UpdateTraining(this.PopulateTrainingVOFromEntryFields(_training_list[_next_training]));
    _training_list = bo.GetTrainingForEmployee(_emp_vo.EmployeeID);
    this.DisplayEmployeeTraining(bo);
  }

  public void DeleteEmployee(Object sender, EventArgs e){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    bo.DeleteEmployee(_emp_vo.EmployeeID);
    _employee_list = bo.GetAllEmployees();
    _next_employee = 0;
    _emp_vo = null;
    this.ResetEntryFields();
    if(_employee_list.Count > 0){
      _emp_vo = _employee_list[_next_employee];
      this.DisplayEmployeeInfo();
      this.DisplayEmployeeTraining(new EmployeeAdminBO());
      if(_training_list.Count > 0){
        _update_training_button.Enabled = true;
        _next_training_button.Enabled = true;
        _delete_training_button.Enabled = true;
      }else{
        _update_training_button.Enabled = false;
        _next_training_button.Enabled = false;
        _delete_training_button.Enabled = false;
      }
      _delete_employee_button.Enabled = true;
    }else{
      _delete_employee_button.Enabled = false;
      _delete_training_button.Enabled = false;
      _next_training_button.Enabled = false;
      _update_training_button.Enabled = false;
      _update_employee_button.Enabled = false;
      _next_employee_button.Enabled = false;
      this.ResetTrainingTextbox();
    }
  }

  public void DeleteTraining(Object sender, EventArgs e){
     EmployeeAdminBO bo = new EmployeeAdminBO();
     bo.DeleteTraining(_training_list[_next_training].TrainingID);
     this.DisplayEmployeeTraining(bo);
     if(_training_list.Count > 0){
        _update_training_button.Enabled = true;
        _next_training_button.Enabled = true;
```

```
      _delete_training_button.Enabled = true;
    } else{
      _update_training_button.Enabled = false;
      _next_training_button.Enabled = false;
      _delete_training_button.Enabled = false;
    }
    _next_training = 0;
    this.ResetTrainingEntryFields();
}

private void AdjustAppWindowSize(){
  this.SuspendLayout();
  _tablepanel.SuspendLayout();
  _employee_info_entry_panel.SuspendLayout();
  _training_info_entry_panel.SuspendLayout();
  _picturebox.Width = _picturebox.Image.Width;
  _picturebox.Height = _picturebox.Image.Height;
  _employee_info_entry_panel.Height = EMPLOYEE_INFO_PANEL_HEIGHT;
  _employee_info_entry_panel.Width = EMPLOYEE_INFO_PANEL_WIDTH;
  _training_info_entry_panel.Height = TRAINING_INFO_PANEL_HEIGHT;
  _training_info_entry_panel.Width = TRAINING_INFO_PANEL_WIDTH;
  _training_textbox.Width = TRAINING_TEXTBOX_WIDTH;
  _training_textbox.Height = TRAINING_TEXTBOX_HEIGHT;
  _tablepanel.Width = (_picturebox.Width + _flowpanel.Width + _employee_info_entry_panel.Width
              + SMALL_PADDING);
  _tablepanel.Height = (_picturebox.Image.Height + _training_textbox.Height + SMALL_PADDING);
  this.Width = _tablepanel.Width + SMALL_PADDING;
  this.Height = _tablepanel.Height;
  _training_info_entry_panel.ResumeLayout();
  _employee_info_entry_panel.ResumeLayout();
  _tablepanel.ResumeLayout();
  this.ResumeLayout();
}

private void DisplayEmployeeTraining(EmployeeAdminBO bo){
  _training_list = bo.GetTrainingForEmployee(_emp_vo.EmployeeID);
  _training_textbox.Text = String.Empty;
  StringBuilder sb = new StringBuilder();
  foreach(TrainingVO t in _training_list){
    sb.Append(t.ToString() + "\r\n");
  }
  _training_textbox.Text = sb.ToString();
}

private TrainingVO.TrainingStatus StringToTrainingStatus(String s){
  TrainingVO.TrainingStatus status = TrainingVO.TrainingStatus.Passed;
  switch(s){
    case "Passed" : status = TrainingVO.TrainingStatus.Passed;
                    break;
    case "Failed" : status = TrainingVO.TrainingStatus.Failed;
                    break;
  }
  return status;
}

private void ResetEntryFields(){
  _fname_textbox.Text = String.Empty;
  _mname_textbox.Text = String.Empty;
  _lname_textbox.Text = String.Empty;
  _male_button.Checked = true;
  _bday_picker.Value = DateTime.Now;
  _picturebox.Image = null;
  this.ResetTrainingEntryFields();
}

private void ResetTrainingEntryFields(){
 _title_textbox.Text = String.Empty;
 _description_textbox.Text = String.Empty;
 _startdate_picker.Value = DateTime.Now;
 _enddate_picker.Value = DateTime.Now;
 _status_listbox.SetSelected(0, true);
}

public void ResetTrainingTextbox(){
  _training_textbox.Text = String.Empty;
}

private void DisplayEmployeeInfo(){
  _fname_textbox.Text = _emp_vo.FirstName;
  _mname_textbox.Text = _emp_vo.MiddleName;
  _lname_textbox.Text = _emp_vo.LastName;
```



```
    switch(_emp_vo.Gender){
      case PersonVO.Sex.MALE : _male_button.Checked = true;
                               break;
      case PersonVO.Sex.FEMALE : _female_button.Checked = true;
                               break;
    }
    _bday_picker.Value = _emp_vo.BirthDay;
    _picturebox.Image = _emp_vo.Picture;
    if(_picturebox.Image != null){
      this.AdjustAppWindowSize();
   }
  }

  private PersonVO.Sex RadioButtonToSexEnum(){
    PersonVO.Sex gender = PersonVO.Sex.MALE;
    if(_male_button.Checked){
      gender = PersonVO.Sex.MALE;
    } else{
      if(_female_button.Checked){
        gender = PersonVO.Sex.FEMALE;
      }
    }
    return gender;
  }

  private EmployeeVO PopulateEmployeeVOFromEntryFields(EmployeeVO vo){
    vo.FirstName = _fname_textbox.Text;
    vo.MiddleName = _mname_textbox.Text;
    vo.LastName = _lname_textbox.Text;
    vo.Gender = this.RadioButtonToSexEnum();
    vo.BirthDay = _bday_picker.Value;
    vo.Picture = _picturebox.Image;
    return vo;
  }

  private TrainingVO PopulateTrainingVOFromEntryFields(TrainingVO vo){
    vo.EmployeeID = _emp_vo.EmployeeID;
    vo.Title = _title_textbox.Text;
    vo.Description = _description_textbox.Text;
    vo.StartDate = _startdate_picker.Value;
    vo.EndDate = _enddate_picker.Value;
    vo.Status = this.StringToTrainingStatus(_status_listbox.SelectedItem.ToString());
    return vo;
  }

  private void DisplayTrainingInfo(TrainingVO vo){
    _title_textbox.Text = vo.Title;
    _description_textbox.Text = vo.Description;
    _startdate_picker.Value = vo.StartDate;
    _enddate_picker.Value = vo.EndDate;
    switch(vo.Status){
      case TrainingVO.TrainingStatus.Passed :
                   _status_listbox.SetSelected(0, true);
                   break;
      case TrainingVO.TrainingStatus.Failed :
                   _status_listbox.SetSelected(1, true);
                   break;
    }
  }

  [STAThread]
  public static void Main(){
    new EmployeeTrainingServer();
  }
}
```

Referring to Example 20.21 — you may be thinking, “Holy cow, you wrote 589 lines of test code?” Trust me, that’s nothing. If you were using a test framework like NUnit to write unit tests for all the individual classes (EmployeeVO, TrainingVO, EmployeeDAO, TrainingDAO, and EmployeeAdminBO), you’d have written more than 588 lines of code, especially if your tests were well thought out and thorough. However, the more effort you put into good unit testing, the easier your programming life becomes, especially when you start to make changes to your code.

The drawbacks to using a GUI application like Example 20.21 to test your code is that it is not automatic. You must make sure to perform all the tasks manually, like creating employees, updating employees, deleting employees, and the same with their associated training records. But it’s better than nothing.

You might also ask, "Why don't you just wait until you build the client to test the code?" That's not a good idea because you really do want to test as you go. You want to move into the client development iteration knowing the server code has been thoroughly tested.

Reality Check

Each development iteration actually comprises many subiterations. For example, the code developed during this second iteration took me about twenty-five subiterations of coding, compiling, and testing.

Third Iteration

At this point the server-side code is nearly complete. All that's left to do is to create the remote object and modify the EmployeeTrainingServer code to host the remote object. This will also require a modification to the Employee-TrainingServer.exe.config file. I will also need to modify the MSBuild project file slightly to add several special build tasks to correctly build the remote object and the EmployeeTrainingServer application. Also, to test the remote object, I'll need to write a short remote client application. Table 20-6 lists the design considerations and design decisions for the third iteration.

Check-Off	Design Consideration	Design Decision
	Remote object interface	Create an interface for the remote object. I'll name the interface IEmployeeTraining. The interface will declare all the methods required to manage employee and training objects.
	Remote object	Create the remote object by extending MarshalByRefObject and implementing the IEmployeeTraining interface. I'll name the remote object EmployeeTrainingRemoteObject.
	EmployeeTrainingServer	Remove the GUI test code and add the code required to host the remote object.
	Configuration file	Add a remoting section.
	Client test application	Start coding the client application. Create a short test application that tests the remote server object. This will require the addition of a client configuration file. The required value object dlls will need to be copied to the client project folder. While I'm at it I'll create an MSBuild project file to help build and manage the client development process.

Table 20-6: Employee Training Server Application — Third Iteration Design Considerations And Decisions

Figure 20-43 shows the UML class diagram for the EmployeeTrainingRemoteObject class. Referring to Figure 20-43 — the EmployeeTrainingRemoteObject class extends MarshalByRefObject and implements the IEmployee-Training interface. It also uses the services of the EmployeeAdminBO class.

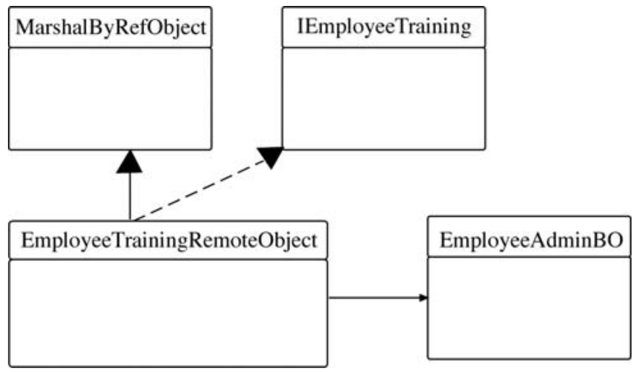


Figure 20-43: EmployeeTrainingRemoteObject UML Class Diagram

Example 20.22 gives the code for the IEmployeeTraining interface.

20.22 IEmployeeTraining.cs

```
using System;
using System.Collections.Generic;
using EmployeeTraining.VO;

public interface IEmployeeTraining {

  #region Employee Methods

  List<EmployeeVO> GetAllEmployees();
  EmployeeVO GetEmployee(Guid employeeID);
  EmployeeVO CreateEmployee(EmployeeVO employee);
  EmployeeVO UpdateEmployee(EmployeeVO employee);
  void DeleteEmployee(Guid employeeID);

  #endregion Employee Methods

  #region Training Methods

  List<TrainingVO> GetTrainingForEmployee(Guid employeeID);
  TrainingVO GetTraining(int trainingID);
  TrainingVO CreateTraining(TrainingVO training);
  TrainingVO UpdateTraining(TrainingVO training);
  void DeleteTraining(int trainingID);
  void DeleteTrainingForEmployee(Guid employeeID);

  #endregion TrainingMethods
}
```

Referring to Example 20.22 — the IEmployeeTraining interface simply declares the methods required to manage employees and their training.

Example 20.23 gives the code for the EmployeeTrainingRemoteObject class.

20.23 EmployeeTrainingRemoteObject.cs

```
using System;
using System.Collections.Generic;
using EmployeeTraining.VO;
using EmployeeTraining.BO;

public class EmployeeTrainingRemoteObject : MarshalByRefObject, IEmployeeTraining {

  #region Employee Methods

  public List<EmployeeVO> GetAllEmployees(){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    return bo.GetAllEmployees();
  }

  public EmployeeVO GetEmployee(Guid employeeID){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    return bo.GetEmployee(employeeID);
  }

  public EmployeeVO CreateEmployee(EmployeeVO employee){
```



```
    EmployeeAdminBO bo = new EmployeeAdminBO();
    return bo.CreateEmployee(employee);
  }

  public EmployeeVO UpdateEmployee(EmployeeVO employee){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    return bo.UpdateEmployee(employee);
  }

  public void DeleteEmployee(Guid employeeID){
    EmployeeAdminBO bo = new EmployeeAdminBO();
    bo.DeleteEmployee(employeeID);
  }

  #endregion Employee Methods

  #region Training Methods

    public TrainingVO CreateTraining(TrainingVO training){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      return bo.CreateTraining(training);
    }

    public TrainingVO GetTraining(int trainingID){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      return bo.GetTraining(trainingID);
    }

    public List<TrainingVO> GetTrainingForEmployee(Guid employeeID){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      return bo.GetTrainingForEmployee(employeeID);
    }

    public TrainingVO UpdateTraining(TrainingVO training){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      return bo.UpdateTraining(training);
    }

    public void DeleteTraining(int trainingID){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      bo.DeleteTraining(trainingID);
    }

    public void DeleteTrainingForEmployee(Guid employeeID){
      EmployeeAdminBO bo = new EmployeeAdminBO();
      bo.DeleteTrainingForEmployee(employeeID);
    }

  #endregion Training Methods
}
```

Referring to Example 20.23 — the EmployeeTrainingRemoteObject class extends MarshalByRefObject and implements the methods required by the IEmployeeTraining interface. In this example, the method implementations simply pass the call on to the corresponding EmployeeAdminBO method.

I did make one change to the EmployeeAdminBO class during this iteration. I modified the DeleteTraining-ForEmployee() method to take a Guid as an argument rather than an EmployeeVO object. This will cut down on network traffic at least somewhat.

Example 20.24 gives the code for the modified EmployeeTrainingServer class.

20.24 EmployeeTrainingServer.cs

```
using System;
using System.Runtime.Remoting;

public class EmployeeTrainingServer {
  public static void Main(){
    RemotingConfiguration.Configure("EmployeeTrainingServer.exe.config", false);
    Console.WriteLine("Listening for remote requests. Press any key to exit...");
    Console.ReadLine();
  }
}
```

Referring to Example 20.24 — this is a whole lot shorter than the last version! This short application simply loads the configuration file. The modified EmployeeTrainingServer.exe.config file is given in Example 20.25.

20.25 EmployeeTrainingServer.exe.config

```
<configuration>
  <configSections>
     <section name="dataConfiguration"
              type="Microsoft.Practices.EnterpriseLibrary.Data.Configuration.DatabaseSettings,
                   Microsoft.Practices.EnterpriseLibrary.Data,
```



```
                    Version=5.0.414.0, Culture=neutral,
                    PublicKeyToken=31bf3856ad364e35" requirePermission="true" />
    </configSections>
  <dataConfiguration defaultDatabase="Connection String" />
  <connectionStrings>
    <add name="Connection String" connectionString="Data Source=(local)\SQLEXPRESS;
                                  Initial Catalog=EmployeeTraining;Integrated Security=True"
                                  providerName="System.Data.SqlClient" />
  </connectionStrings>
  <system.runtime.remoting>
    <application>
      <service>
         <wellknown mode="Singleton"
              type="EmployeeTrainingRemoteObject, EmployeeTrainingRemoteObject"
              objectUri="EmployeeTraining" />
      </service>
      <channels>
        <channel ref="tcp" port="8080" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Referring to Example 20.25 — the configuration file now sports a `<system.runtime.remoting>` section which gives configuration details about the remote object, its hosting mode (Singleton), and its URI.

Now, to compile the IEmployeeTraining interface, the EmployeeTrainingRemoteObject class, and the EmployeeTrainingServer class, you'll need to make a modification to the MSBuild project file. The modified project file is listed in Example 20.26.

20.26 EmployeeTrainingServer.proj (Mod 2)

```
<Project DefaultTargets="CompileApp"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

   <PropertyGroup>
     <IncludeDebugInformation>false</IncludeDebugInformation>
     <BuildDir>build</BuildDir>
     <LibDir>lib</LibDir>
     <AppDir>app</AppDir>
     <RefDir>ref</RefDir>
     <ConfigDir>config</ConfigDir>
   </PropertyGroup>

    <ItemGroup>
      <DAO Include="dao\**\*.cs" />
      <BO Include="bo\**\*.cs" />
      <VO Include="vo\**\*.cs" />
      <APP Include="app\EmployeeTrainingServer.cs" />
      <REMOTEINTERFACE Include="app\IEmployeeTraining.cs" />
      <REMOTEOBJECT Include="app\EmployeeTrainingRemoteObject.cs" />
      <LIB Include="lib\**\*.dll" />
      <REF Include="ref\**\*.dll" />
      <CONFIG Include="config\**\*.config" />
      <EXE Include="app\**\*.exe" />
    </ItemGroup>

    <Target Name="MakeDirs">
      <MakeDir Directories="$(BuildDir)" />
      <MakeDir Directories="$(LibDir)" />
    </Target>

    <Target Name="RemoveDirs">
      <RemoveDir Directories="$(BuildDir)" />
      <RemoveDir Directories="$(LibDir)" />
    </Target>

    <Target Name="Clean"
            DependsOnTargets="RemoveDirs;MakeDirs">
    </Target>

     <Target Name="CopyFiles">
       <Copy
         SourceFiles="@(CONFIG);@(LIB);@(REF)"
         DestinationFolder="$(BuildDir)" />
    </Target>

    <Target Name="CompileVO"
            Inputs="@(VO)"
            Outputs="$(LibDir)\VOLib.dll">
      <Csc Sources="@(VO)"
           TargetType="library"
```



```
          References="@(REF);@(LIB)"
          OutputAssembly="$(LibDir)\VOLib.dll">
      </Csc>
   </Target>

    <Target Name="CompileDAO"
            Inputs="@(DAO)"
            Outputs="$(LibDir)\DAOLib.dll"
            DependsOnTargets="CompileVO">
       <Csc Sources="@(DAO)"
          TargetType="library"
          References="@(REF);@(LIB)"
          WarningLevel="0"
          OutputAssembly="$(LibDir)\DAOLib.dll">
      </Csc>
    </Target>

    <Target Name="CompileBO"
            Inputs="@(BO)"
            Outputs="$(LibDir)\BOLib.dll"
            DependsOnTargets="CompileDAO">
      <Csc Sources="@(BO)"
          TargetType="library"
          References="@(REF);@(LIB)"
          WarningLevel="0"
          OutputAssembly="$(LibDir)\BOLib.dll">
      </Csc>
    </Target>

    <Target Name="CompileApp"
            Inputs="@(APP);@(REMOTEINTERFACE);@(REMOTEOBJECT)"
            Outputs="$(BuildDir)\$(MSBuildProjectName).exe;
                     $(LibDir)\IEmployeeTraining.dll;
                     $(LibDir)\EmployeeTrainingRemoteObject.dll"
            DependsOnTargets="CompileBO">
      <Csc Sources="@(REMOTEINTERFACE)"
           TargetType="library"
           References="@(REF);@(LIB)"
           OutputAssembly="$(LibDir)\IEmployeeTraining.dll">
       </Csc>
       <Csc Sources="@(REMOTEOBJECT)"
           TargetType="library"
           References="@(REF);@(LIB)"
           OutputAssembly="$(LibDir)\EmployeeTrainingRemoteObject.dll">
       </Csc>
      <Csc Sources="@(APP)"
           TargetType="exe"
           References="@(REF);@(LIB)"
           OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
      </Csc>
    </Target>

    <Target Name="Run"
            DependsOnTargets="CompileApp;CopyFiles">
      <Exec Command="$(MSBuildProjectName).exe"
            WorkingDirectory="$(BuildDir)" />
    </Target>

</Project>
```

Referring to Example 20.26 — I've made changes to the <ItemGroup> section and to the <CompileApp> target. To the <ItemGroup> section I added <REMOTEINTERFACE> and <REMOTEOBJECT> items, giving specific names for the corresponding source files. To the <CompileApp> target I added two new <Csc> tasks to compile the IEmployeeTraining and EmployeeTrainingRemoteObject source files.

To compile the EmployeeTrainingServer application, simply execute the CompileApp target by entering the following command-line command:

```
msbuild /t:compileapp
```

If all goes well the EmployeeTrainingServer.exe file will be built and written to the build directory. Change to the build directory and double-click the EmployeeTrainingServer.exe file. You should see an output similar to that shown in Figure 20-44.

To test the server at this point requires building a suitable remoting client application. I cover this topic in the next section.

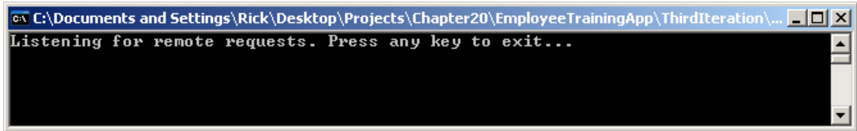


Figure 20-44: EmployeeTrainingServer Running and Ready For Remote Connections

The Client Application

In this section I will show you how to build a suitable remoting client application that provides a GUI front-end to the EmployeeTrainingServer application. The GUI-based client application will allow users to manage employees and their training with the help of menus, dialog boxes, and data grid components.

Third Iteration (continued)

The best place to start the client development effort is by setting up the client project folders, building an MSbuild project file, creating a client configuration file, and writing a small client application to test connectivity to the EmployeeTrainingRemoteObject. Table 20-7 lists the design considerations and design decisions for the continuing third iteration.

Check-Off	Design Consideration	Design Decision
	Project directory structure	Create the client application project folders. In the client directory create the app, build, config, and ref subdirectories.
	MSBuild project file	Create an MSBuild project file that will be used to compile and run the client application.
	Client configuration file	Create a configuration file that contains a <system.runtime.remoting> section. The name of the configuration file will be EmployeeTraining-Client.exe.config
	Remoting client application	Start the client application by writing a short program that tests the connection to the remote object.

Table 20-7: Employee Training Client Application — Third Iteration Design Considerations And Decisions (Continued)

Figure 20-45 shows the client project directory structure.

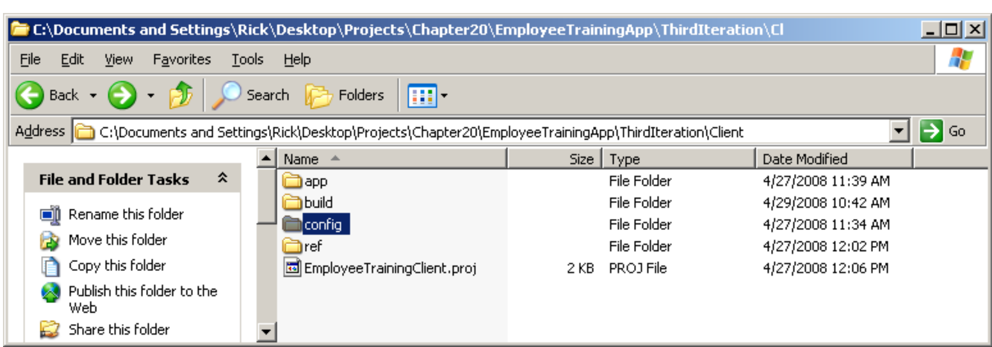


Figure 20-45: Client Project Directory Structure

Referring to Figure 20-45 — the client application source code goes in the app folder. The configuration file resides in the config folder, and the required dlls must be placed in the ref folder. For this iteration you will need the IEmployeeTraining.dll and the VOLib.dll files. You will find these dlls in the server project's lib directory. The client executable file will be built to the build folder and any required dlls will be moved to that location as well.

Example 20.27 gives the code for the EmployeeTrainingClient.proj project file.

20.27 EmployeeTrainingClient.proj

```
1   <Project DefaultTargets="Run"
2            xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
3
4      <PropertyGroup>
5        <IncludeDebugInformation>false</IncludeDebugInformation>
6        <BuildDir>build</BuildDir>
7        <AppDir>app</AppDir>
8        <RefDir>ref</RefDir>
9        <ConfigDir>config</ConfigDir>
10     </PropertyGroup>
11
12      <ItemGroup>
13
14        <APP Include="app\EmployeeTrainingClient.cs" />
15        <REF Include="ref\**\*.dll" />
16        <CONFIG Include="config\**\*.config" />
17        <EXE Include="app\**\*.exe" />
18      </ItemGroup>
19
20      <Target Name="MakeDirs">
21        <MakeDir Directories="$(BuildDir)" />
22      </Target>
23
24      <Target Name="RemoveDirs">
25        <RemoveDir Directories="$(BuildDir)" />
26      </Target>
27
28      <Target Name="Clean"
29              DependsOnTargets="RemoveDirs;MakeDirs">
30      </Target>
31
32       <Target Name="CopyFiles">
33         <Copy
34           SourceFiles="@(CONFIG);@(REF)"
35           DestinationFolder="$(BuildDir)" />
36      </Target>
37
38      <Target Name="CompileApp"
39              Inputs="@(APP)"
40              Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
41              DependsOnTargets="Clean">
42        <Csc Sources="@(APP)"
43             TargetType="exe"
44             References="@(REF)"
45             OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
46        </Csc>
47      </Target>
48
49      <Target Name="Run"
50              DependsOnTargets="CompileApp;CopyFiles">
51        <Exec Command="$(MSBuildProjectName).exe"
52              WorkingDirectory="$(BuildDir)" />
53      </Target>
54  </Project>
```

Referring to Example 20.27 — this project file contains <PropertyGroup> and <ItemGroup> sections along with several targets. There are two primary targets: CompileApp and Run. The default project target is specified on line 1 as the Run target. The Run target depends on the CompileApp and CopyFiles targets.

Example 20.28 gives the code for the EmployeeTrainingClient.exe.config configuration file.

20.28 EmployeeTrainingClient.exe.config

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="IEmployeeTraining, IEmployeeTraining"
            url="tcp://localhost:8080/EmployeeTraining" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```


Referring to Example 20.28 — the client configuration file has a <system.runtime.remoting> section, which specifies the remote object type and its url.

Example 20.29 gives the code for the EmployeeTrainingClient application.

20.29 EmployeeTrainingClient.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using EmployeeTraining.VO;

public class EmployeeTrainingClient {
  public static void Main(){
   try {
     RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
     WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
     IEmployeeTraining employee_training =
      (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
     Console.WriteLine("Remote EmployeeTraining object successfully created!");
     List<EmployeeVO> employee_list = employee_training.GetAllEmployees();
     foreach(EmployeeVO emp in employee_list){
       Console.WriteLine(emp.FirstName + " " + emp.MiddleName + " " + emp.LastName);
     }
    }catch(Exception e){
      Console.WriteLine(e);
    }
  }
}
```

Referring to Example 20.29 — this first short version of the client application tests the connectivity to the remote object. Once it obtains the proxy to the remote object, it calls the GetAllEmployees() method and prints the returned information to the console.

To build and run this application make sure you've copied the required dlls to the client's ref folder and have started the server. Run the msbuild project file's Run target with the following command-line command:

```
msbuild /t:run
```

Also, since the Run target is the default target, you could also simply enter the following command:

```
msbuild
```

Figure 20-46 shows the results of running the first version of the client application.

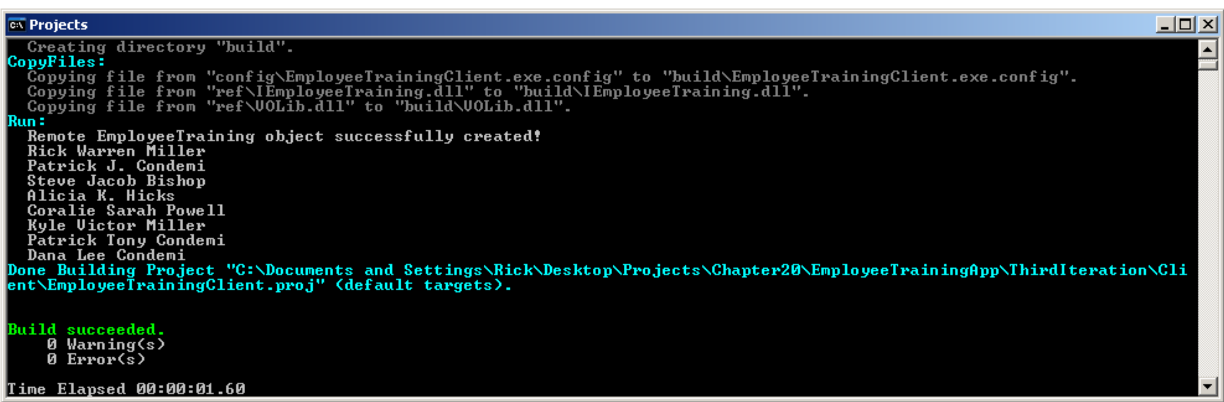


Figure 20-46: Running Client Application via the MSBuild Project's Run Target

Fourth Iteration

It's time now in this development iteration to flesh out the final version of the Employee Training client application. As you proceed with development you may find that you'll need to make some changes to the server application in order to accommodate some unforeseen design problems.

A good place to start this development cycle is to sketch out a framework for the client GUI application, implement a piece of it, and continue with testing the server application, as the minimal amount of testing done in the pre-

vious iteration was wholly inadequate. Table 20-8 lists the design considerations and design decisions for the fourth development iteration.

Check-Off	Design Consideration	Design Decision
	Client application	Sketch out a mock-up of the client application GUI and start its implementation. The client application will need to use the EmployeeTraining remote object, so you'll need to pass a reference to the remote object into the client application. This you can do via the client application constructor.
	Application testing	Continue testing the server side components and note any deficiencies.

Table 20-8: Employee Training Client Application — Fourth Iteration Design Considerations And Decisions

Referring to Table 20-8 — these two activities are quite enough to bite off for this iteration. Let's start with a UML diagram of the EmployeeTrainingClient application class, as is shown in Figure 20-47.

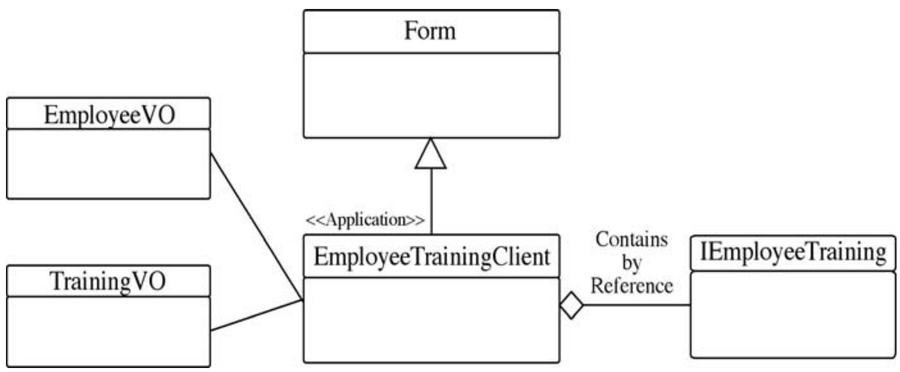


Figure 20-47: EmployeeTrainingClient UML Class Diagram

Referring to Figure 20-47 — the EmployeeTrainingClient class extends the Form class. (System.Windows.Form) It also contains by reference an instance of IEmployeeTraining, and it has a dependency on the EmployeeVO and TrainingVO classes. Thus, if changes are required to the server side components, you'll need to ensure you copy the required dependant dlls into the client project's ref folder before building the client application. The dependent dlls include VOLib.dll and IEmployeeTraining.dll.

Figure 20-48 shows a mock-up sketch of the GUI layout for the EmployeeTrainingClient application.

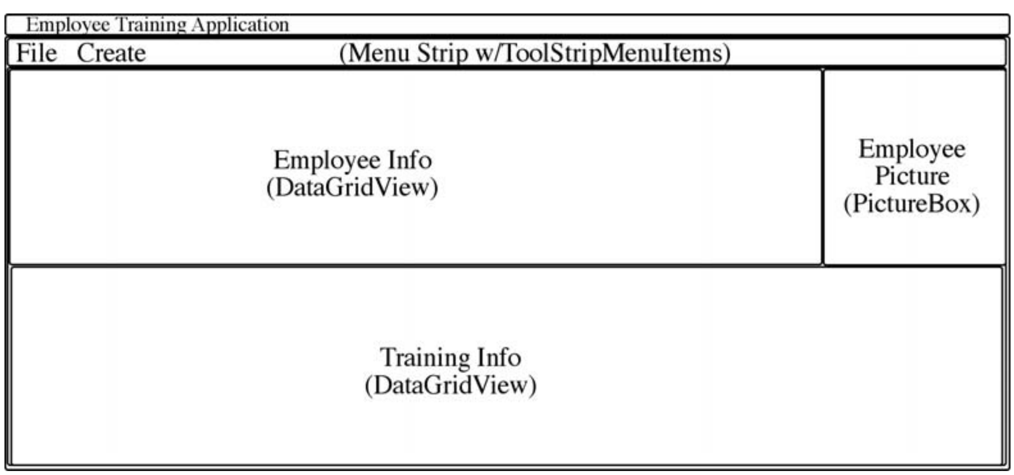


Figure 20-48: Mock-up Sketch of the EmployeeTrainingApplication GUI

Referring to Figure 20-48 — the GUI contains a menu with several menu items. Here I've only shown two menu items, but the final application may contain more. DataGridView components are used to display employee and training information. A PictureBox component contains the employee's picture. The components are arranged in a TableLayoutPanel containing two rows and two columns. The employee DataGridView goes into the upper left table layout cell, and the PictureBox is placed in the upper right cell. The training DataGridView is placed in the second row and spans two columns. Example 20.30 gives the code for the partial implementation of this application.

20.30 EmployeeTrainingClient.cs

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.IO;
using System.ComponentModel;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using EmployeeTraining.VO;

public class EmployeeTrainingClient : Form {

  // Constants
  private const int WINDOW_HEIGHT = 500;
  private const int WINDOW_WIDTH = 900;
  private const String WINDOW_TITLE = "Employee Training Application";
  private const bool DEBUG = true;

  // fields
  private IEmployeeTraining _employeeTraining = null;
  private List<EmployeeVO>  _employeeList = null;
  private TableLayoutPanel _tablePanel = null;
  private DataGridView _employeeGrid = null;
  private DataGridView _trainingGrid = null;
  private PictureBox _pictureBox = null;

  public EmployeeTrainingClient(IEmployeeTraining employeeTraining){
    _employeeTraining = employeeTraining;
    this.InitializeComponent();
  }

  private void InitializeComponent(){
    // setup the menus
    MenuStrip ms = new MenuStrip();

    ToolStripMenuItem fileMenu = new ToolStripMenuItem("File");
    ToolStripMenuItem exitMenuItem = new ToolStripMenuItem("Exit", null,
                                      new EventHandler(this.ExitProgramHandler));

    ToolStripMenuItem createMenu = new ToolStripMenuItem("Create");
    ToolStripMenuItem employeeMenuItem = new ToolStripMenuItem("Employee...", null,
                                    new EventHandler(this.CreateEmployeeHandler));
    ToolStripMenuItem trainingMenuItem = new ToolStripMenuItem("Training...", null,
                                    new EventHandler(this.CreateTrainingHandler));

    fileMenu.DropDownItems.Add(exitMenuItem);
    ms.Items.Add(fileMenu);

    createMenu.DropDownItems.Add(employeeMenuItem);
    createMenu.DropDownItems.Add(trainingMenuItem);
    ms.Items.Add(createMenu);

    // create the table panel
    _tablePanel = new TableLayoutPanel();
    _tablePanel.RowCount = 2;
    _tablePanel.ColumnCount = 2;
    _tablePanel.Anchor =  AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
    _tablePanel.Dock = DockStyle.Top;
    _tablePanel.Height = 400;

    // create and initialize the data grids
    _employeeGrid = new DataGridView();
    _employeeGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    _employeeGrid.Height = 200;
    _employeeGrid.Width = 700;
    _employeeList = _employeeTraining.GetAllEmployees();
    _employeeGrid.DataSource = _employeeList;
    _employeeGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
```



```
70       _employeeGrid.Click += this.EmployeeGridClickedHandler;
71
72       _trainingGrid = new DataGridView();
73       _trainingGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
74       _trainingGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
75
76       // create picture box
77       _pictureBox = new PictureBox();
78       _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
79
80       //add grids to table panel
81       _tablePanel.Controls.Add(_employeeGrid);
82       _tablePanel.Controls.Add(_pictureBox);
83       _tablePanel.Controls.Add(_trainingGrid);
84       _tablePanel.SetColumnSpan(_trainingGrid, 2);
85
86       this.Controls.Add(_tablePanel);
87       ms.Dock = DockStyle.Top;
88       this.MainMenuStrip = ms;
89       this.Controls.Add(ms);
90       this.Height = WINDOW_HEIGHT;
91       this.Width = WINDOW_WIDTH;
92       this.Text = WINDOW_TITLE;
93     }
94
95     /************************************************************
96       Event Handlers
97     ************************************************************/
98     private void ExitProgramHandler(Object sender, EventArgs e){
99       Application.Exit();
100    }
101
102    private void CreateEmployeeHandler(Object sender, EventArgs e){
103      // add code here
104    }
105
106    private void CreateTrainingHandler(Object sender, EventArgs e){
107      // add code here
108    }
109
110    private void EmployeeGridClickedHandler(Object sender, EventArgs e){
111      int selected_row = _employeeGrid.SelectedRows[0].Index;
112      Image employee_picture = _employeeList[selected_row].Picture;
113
114      if(employee_picture != null){
115        _pictureBox.Image = employee_picture;
116      }
117      if(DEBUG){ // print some info to the console
118        Console.WriteLine(selected_row);
119        Console.WriteLine(_employeeList[selected_row]);
120      }
121
122      _trainingGrid.DataSource = null;
123      _trainingGrid.DataSource =
124               _employeeTraining.GetTrainingForEmployee(_employeeList[selected_row].EmployeeID);
125    }
126
127    [STAThread]
128    public static void Main(){
129     try {
130       RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
131       WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
132       IEmployeeTraining employee_training =
133        (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
134       EmployeeTrainingClient client = new EmployeeTrainingClient(employee_training);
135       Application.Run(client);
136      }catch(Exception e){
137        Console.WriteLine(e);
138      }
139    }
140  } // end class definition
```

Referring to Example 20.30 — the EmployeeTrainingClient class extends Form, as expected. It contains two DataGridViews and a PictureBox, which are contained within a TableLayoutPanel in accordance with the mock-up sketch given in Figure 20-48. All menu item event handler methods, with the exception of the File->Exit menu item event handler, are stub methods that will eventually need to be fleshed out.

Let's take a look at the Main() method which begins on line 128. The bulk of the Main() method remains unchanged from the previous iteration. The test code has been removed and replaced with lines 133 and 134. These

lines of code create an instance of the EmployeeTrainingClient, passing into the constructor the reference to the remote object, and then calling Application.Run() to kick things off.

Look now at the InitializeComponent() method which begins on line 33. The first thing I do is create and initialize the menu strip and its associated menu items. Next, beginning on line 55, I create and initialize the TableLayoutPanel, followed by the creation and initialization of the DataGridView components. When I create the _employeeGrid, I make a call via the remote object reference _employeeTraining to get a list of all employees. I assign this list to the _employeeList reference and then use this reference to set the _employeeGrid.DataSource property.

So, what happens when the application starts is this: The client application displays a list of employees and their associated data in the employee DataGridView component. When a user clicks on the employee DataGridView, that Click event is handled by the EmployeeGridClickedHandler() method, which begins on line 110. The event handler loads the employee's picture into the PictureBox component as long as the employee's picture is not null. It then loads the employee's training into the training DataGridView by setting its DataSource property via a call to the remote object's GetTrainingForEmployee() method. A click on a DataGridView yields a row index value. This row index value is used to index the _employeeList to retrieve the appropriate EmployeeVO object.

To compile and run this application make sure you've copied the requisite dlls from the server application's lib folder to the client application's ref folder, start the server application, and then from the client application project directory run MSBuild with the default target like so:

```
msbuild
```

If all goes well you'll see the client application window open and it should look something like Figure 20-49.

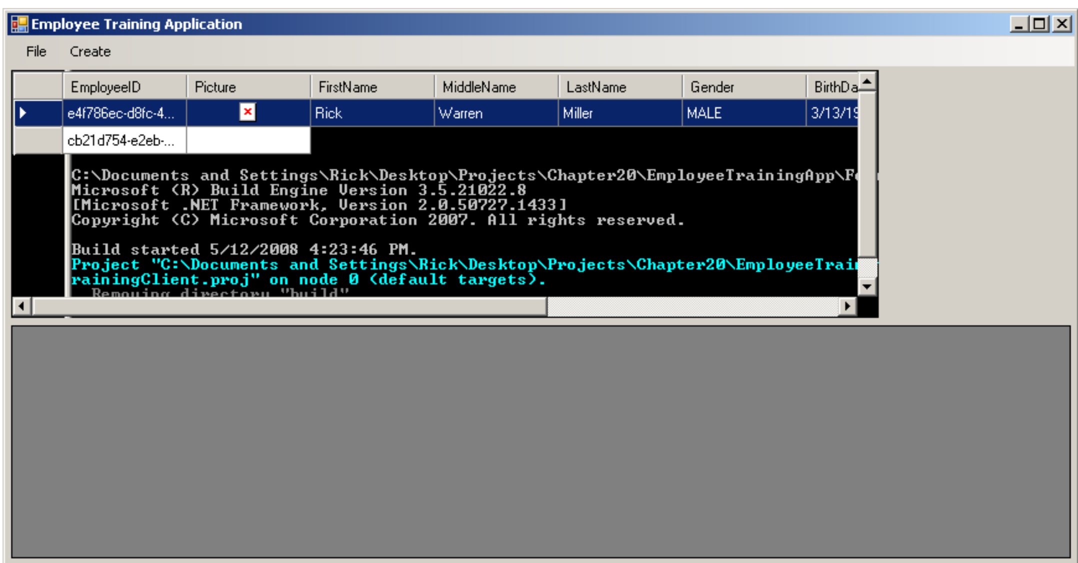


Figure 20-49: EmployeeTrainingClient Initial Display on Startup — Something's Not Quite Right!

Referring to Figure 20-49 — well, something's amiss! Your display will look different depending on what was behind your application window on startup. Let's try clicking on the first row of the employee information DataGridView and see if the related training will display. Figure 20-50 shows the results.

Referring to Figure 20-50 — when I click the first row (I'm clicking on the gray margin to the left of each row) the related training for that employee shows up in the training DataGridView. However, the first employee has no picture. Let's see what happens when I click on the second row. Figure 20-51 shows the results. (Cross your fingers!)

Referring to Figure 20-51 — something is obviously wrong! I've received a rather cryptic RemotingException saying: "Remoting cannot find field 'nativeImage' on type System.Drawing.Image." Upon deep investigation around the Internet I finally find the following note buried on the MSDN website for the System.Drawing.Bitmap class:

> *"The Bitmap class is not accessible across application domains. For example, if you create a dynamic AppDomain and create several brushes, pens, and bitmaps in that domain, then pass these objects back to the main application domain, you can successfully use the pens and brushes. However, if you call the DrawImage method to draw the marshaled Bitmap, you receive the following exception. Remoting cannot find field "native image" on type "System.Drawing.Image". "*

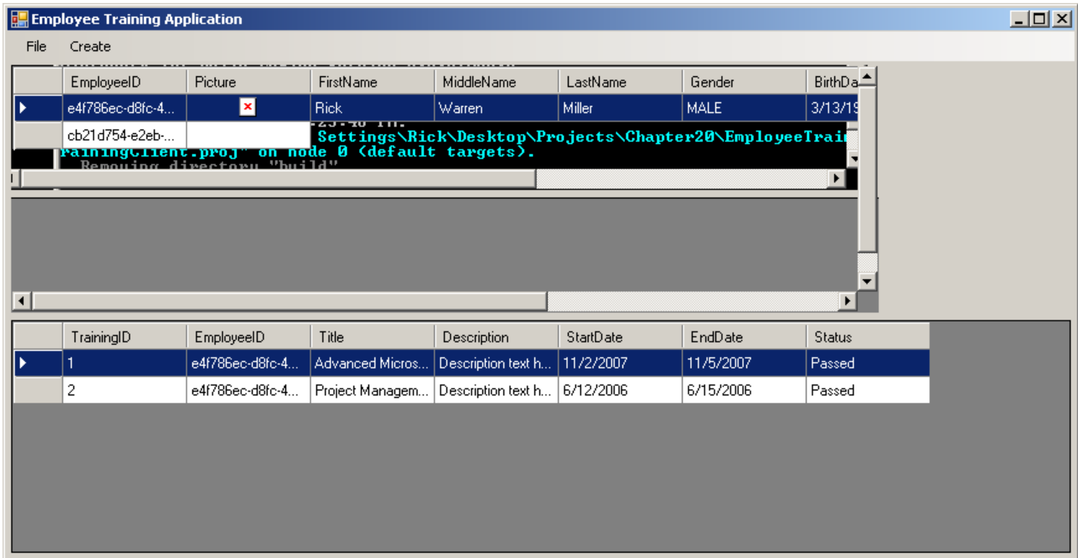


Figure 20-50: Employee’s Related Training Shown in Training DataGridView

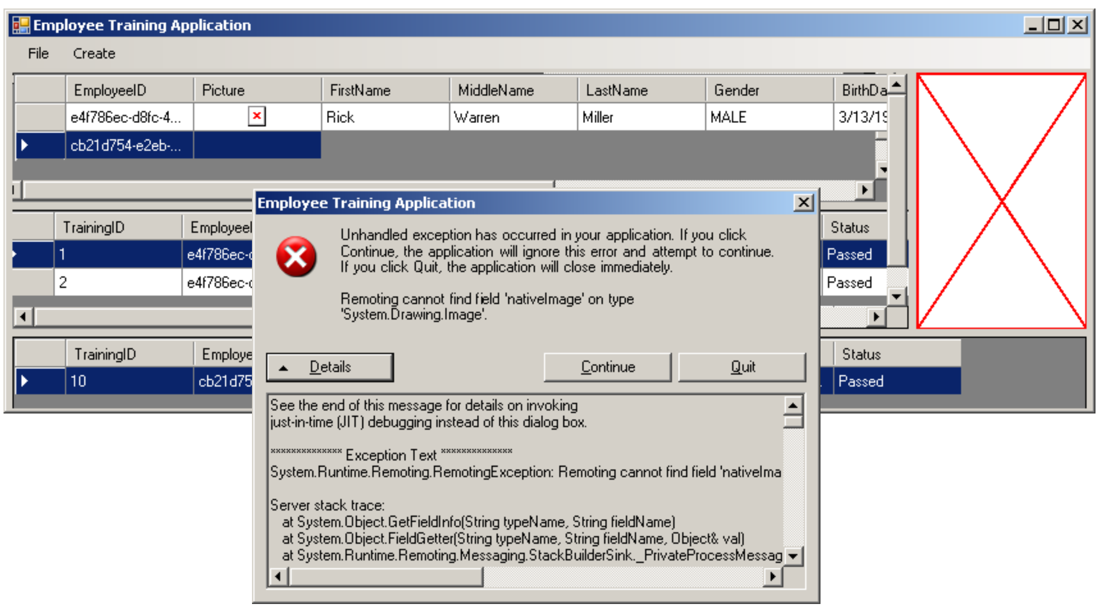


Figure 20-51: Results of Clicking on a Employee with a Picture - a RemotingException is Thrown

Note:

The Bitmap class is not accessible across application domains. For example, if you create a dynamic AppDomain and create several brushes, pens, and bitmaps in that domain, then pass these objects back to the main application domain, you can successfully use the pens and brushes. However, if you call the DrawImage method to draw the marshaled Bitmap, you receive the following exception.

Remoting cannot find field "native image" on type "System.Drawing.Image".

Figure 20-52: Bitmap Class Usage Note

I’ve also shown the Bitmap note in Figure 20-52. OK, so if you can’t transfer an Image across application domains, how are you to transfer the employee’s picture? You’ll have to do it the old fashioned way — store the

employee's picture as an array of bytes. These should transfer across application domains with no problem. To do this will require some changes to the server application. This fix will be the focus of the fifth development iteration.

Fifth Iteration

In the previous development iteration we encountered a problem with transferring the Employee's picture across application domains via .NET remoting. This problem played havoc with the employee DataGridView component. In this iteration I'm going to fix that problem by modifying the server application to hold the employee's picture as an array of bytes. (*i.e.,* a byte[]) To make this fix I'll need to modify two server-side classes: EmployeeVO and EmployeeDAO. I'll also need to modify the EmployeeTrainingClient class to properly handle the modified EmployeeVO class. (You see, it's sweet having an application architecture that lets you zero in on exactly what components need to be modified to implement the fix.)

Table 20-9 gives the design considerations and design decisions for the fifth iteration.

Check-Off	Design Consideration	Design Decision
	Employee picture transfer problem: EmployeeVO class	Modify the EmployeeVO class to hold employee picture data in a byte array.
	Employee picture transfer problem: EmployeeDAO class	Modify the EmployeeDAO class to properly insert the byte array into the tbl_employee.Picture column and to properly populate the EmployeeVO upon retrieval.
	EmployeeTrainingClient class	Modify the EmployeeGridClickedHandler() method to properly handle the modified EmployeeVO class.
	Application testing	Continue with application testing to ensure the changes work. Some of the changes to the DAO will not be tested fully until the next iteration.

Table 20-9: Employee Training Client Application — Fifth Iteration Design Considerations And Decisions

Example 20.31 gives the modified code for the EmployeeVO class.

20.31 EmployeeVO.cs (modified)

```
using System;

namespace EmployeeTraining.VO {
[Serializable]
  public class EmployeeVO : PersonVO {

  // private instance fields
  private Guid     _employeeID;
  private byte[] _picturebytes;

  //default constructor
  public EmployeeVO(){}

  public EmployeeVO(Guid employeeid, String firstName, String middleName, String lastName,
              Sex gender, DateTime birthday):base(firstName, middleName, lastName, gender, birthday){
     EmployeeID = employeeid;
  }

  // public properties
  public Guid EmployeeID {
    get { return _employeeID; }
    set { _employeeID = value; }
  }

  public byte[] Picture {
    get { return _picturebytes; }
    set { _picturebytes = value; }
  }

  public override String ToString(){
    return (EmployeeID + " " + base.ToString());
  }
} // end EmployeeVO class
} // end namespace
```

Referring to Example 20.31 — I’ve made three changes to this class. First, I removed the `using` System.Drawing directive since I no longer need to use the System.Drawing.Image class. Second, I removed the _picture field and replaced it with the _picturebytes field which is of type byte array (byte[]). Lastly, I changed the Picture property to reflect it’s new type and to get and set the _picturebytes field.

Example 20.32 gives the code for the modified EmployeeDAO class.

20.32 EmployeeDAO.cs (modified)

```
using System;
using System.IO;
using System.Data;
using System.Data.Common;
using System.Data.Sql;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using System.Collections.Generic;
//using System.Drawing;
//using System.Drawing.Imaging;
using EmployeeTraining.VO;

using Microsoft.Practices.EnterpriseLibrary.Common;
using Microsoft.Practices.EnterpriseLibrary.Data;
using Microsoft.Practices.EnterpriseLibrary.Data.Sql;

namespace EmployeeTraining.DAO {
  public class EmployeeDAO : BaseDAO {

   private bool debug = true;

    //List of column identifiers used in perpared statements
    private const String EMPLOYEE_ID = "@employee_id";
    private const String FIRST_NAME = "@first_name";
    private const String MIDDLE_NAME = "@middle_name";
    private const String LAST_NAME = "@last_name";
    private const String BIRTHDAY = "@birthday";
    private const String GENDER = "@gender";
    private const String PICTURE = "@picture";

    private const String SELECT_ALL_COLUMNS =
      "SELECT employeeid, firstname, middlename, lastname, birthday, gender, picture ";

    private const String SELECT_ALL_EMPLOYEES =
      SELECT_ALL_COLUMNS +
      "FROM tbl_employee ";

    private const String SELECT_EMPLOYEE_BY_EMPLOYEE_ID =
      SELECT_ALL_EMPLOYEES +
      "WHERE employeeid = " + EMPLOYEE_ID;


    private const String INSERT_EMPLOYEE =
      "INSERT INTO tbl_employee " +
          "(EmployeeID, FirstName, MiddleName, LastName, Birthday, Gender, Picture) " +
       "VALUES (" + EMPLOYEE_ID + ", " + FIRST_NAME + ", " + MIDDLE_NAME + ", " + LAST_NAME + ", " +
                    BIRTHDAY + ", " + GENDER + ", " + PICTURE + ")";

    private const String UPDATE_EMPLOYEE =
      "UPDATE tbl_employee " +
      "SET FirstName = " + FIRST_NAME + ", MiddleName = " + MIDDLE_NAME + ", LastName = " + LAST_NAME +
             ", Birthday = " + BIRTHDAY + ", Gender = " + GENDER + ", Picture = " + PICTURE + " " +
      "WHERE EmployeeID = " + EMPLOYEE_ID;

    private const String DELETE_EMPLOYEE =
      "DELETE FROM tbl_employee " +
      "WHERE EmployeeID = " + EMPLOYEE_ID;

    /************************************
              Returns a List<EmployeeVO> object
          *************************************/
    public List<EmployeeVO> GetAllEmployees(){
      DbCommand command = DataBase.GetSqlStringCommand(SELECT_ALL_EMPLOYEES);
      return this.GetEmployeeList(command);
    }

    /***********************************************
             Returns an EmployeeVO object given a valid employeeid
          ***********************************************/
    public EmployeeVO GetEmployee(Guid employeeid){
      DbCommand command = null;
      try{
```



```
        command = DataBase.GetSqlStringCommand(SELECT_EMPLOYEE_BY_EMPLOYEE_ID);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetEmployee(command);
    }

    /*******************************************************
            Inserts an employee given a fully-populated EmployeeVO object
        ********************************************************/
    public EmployeeVO InsertEmployee(EmployeeVO employee){
      try{
        employee.EmployeeID = Guid.NewGuid();
        DbCommand command = DataBase.GetSqlStringCommand(INSERT_EMPLOYEE);
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
        DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
        DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
        DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
        DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
        switch(employee.Gender){
          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
               break;
          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
               break;
        }

        if(employee.Picture != null){
         if(debug){  Console.WriteLine("Inserting picture!"); }
          if(debug){
            for(int i=0; i<employee.Picture.Length; i++){
              Console.Write(employee.Picture[ i] );
            }
          }  // end if debug
         DataBase.AddInParameter(command, PICTURE, DbType.Binary, employee.Picture);
         if(debug){  Console.WriteLine("Picture inserted, I think!"); }
        }
        DataBase.ExecuteNonQuery(command);
      } catch(Exception e){
        Console.WriteLine(e);
      }
      return this.GetEmployee(employee.EmployeeID);
    }

    /********************************************************
             Updates a row in the tbl_employee table given the fully-populated
             EmployeeVO object.
         *********************************************************/
    public EmployeeVO UpdateEmployee(EmployeeVO employee){
      try {
        DbCommand command = DataBase.GetSqlStringCommand(UPDATE_EMPLOYEE);
        DataBase.AddInParameter(command, FIRST_NAME, DbType.String, employee.FirstName);
        DataBase.AddInParameter(command, MIDDLE_NAME, DbType.String, employee.MiddleName);
        DataBase.AddInParameter(command, LAST_NAME, DbType.String, employee.LastName);
        DataBase.AddInParameter(command, BIRTHDAY, DbType.DateTime, employee.BirthDay);
        switch(employee.Gender){
          case EmployeeVO.Sex.MALE: DataBase.AddInParameter(command, GENDER, DbType.String, "M");
               break;
          case EmployeeVO.Sex.FEMALE: DataBase.AddInParameter(command, GENDER, DbType.String, "F");
               break;
        }
        if(employee.Picture != null){
         if(debug){  Console.WriteLine("Inserting picture!"); }
          if(debug){
            for(int i=0; i<employee.Picture.Length; i++){
              Console.Write(employee.Picture[ i] );
            }
          }  // end if debug
         DataBase.AddInParameter(command, PICTURE, DbType.Binary, employee.Picture);
         if(debug){  Console.WriteLine("Picture inserted, I think!"); }
        }
        DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employee.EmployeeID);
        DataBase.ExecuteNonQuery(command);
    } catch(Exception e){
      Console.WriteLine(e);
    }
    return this.GetEmployee(employee.EmployeeID);
    }

   /*******************************************************
           Deletes a row from the tbl_employee table given an employee id.
```

```
      ********************************************************/
   public void DeleteEmployee(Guid employeeid){
     try{
       DbCommand command = DataBase.GetSqlStringCommand(DELETE_EMPLOYEE);
       DataBase.AddInParameter(command, EMPLOYEE_ID, DbType.Guid, employeeid);
       DataBase.ExecuteNonQuery(command);
     }catch(Exception e){
       Console.WriteLine(e);
     }
   }

    /***********************************************
              Private utility method that executes the given DbCommand
              and returns a fully-populated EmployeeVO object
        ***********************************************/
    private EmployeeVO GetEmployee(DbCommand command){
      EmployeeVO empVO = null;
      IDataReader reader = null;
      try {
        reader = DataBase.ExecuteReader(command);
        if(reader.Read()){
          empVO = this.FillInEmployeeVO(reader);
        }
      }catch(Exception e){
        Console.WriteLine(e);
      }finally {
        base.CloseReader(reader);
      }
      return empVO;
    }

    /***************************************************
             GetEmployeeList() - returns a List<EmployeeVO> object
        ***************************************************/
    private List<EmployeeVO> GetEmployeeList(DbCommand command){
      IDataReader reader = null;
      List<EmployeeVO> employee_list = new List<EmployeeVO>();
      try{
        reader = DataBase.ExecuteReader(command);
        while(reader.Read()){
          EmployeeVO empVO = this.FillInEmployeeVO(reader);
          employee_list.Add(empVO);
        }
      }catch(Exception e){
        Console.WriteLine(e);
      }finally{
        base.CloseReader(reader);
      }
      return employee_list;
    }

    /*******************************************************
             Private utility method that populates an EmployeeVO object from
             data read from the IDataReader object
       *********************************************************/
    private EmployeeVO FillInEmployeeVO(IDataReader reader){
      EmployeeVO empVO = new EmployeeVO();
      empVO.EmployeeID = reader.GetGuid(0);
      empVO.FirstName = reader.GetString(1);
      empVO.MiddleName = reader.GetString(2);
      empVO.LastName = reader.GetString(3);
      empVO.BirthDay = reader.GetDateTime(4);
      String gender = reader.GetString(5);
      switch(gender){
        case "M" : empVO.Gender = EmployeeVO.Sex.MALE;
                   break;
        case "F" : empVO.Gender = EmployeeVO.Sex.FEMALE;
                   break;
      }
      if(!reader.IsDBNull(6)){
        int buffersize = 5000;
        int startindex = 0;
        Byte[] byte_array = new Byte[buffersize];
        MemoryStream ms = new MemoryStream();
        long retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
        while(retval > 0){
          ms.Write(byte_array, 0, byte_array.Length);
          startindex += buffersize;
          retval = reader.GetBytes(6, startindex, byte_array, 0, buffersize);
         }
        empVO.Picture = ms.ToArray();
```



```
      }
      return empVO;
    }

  } // end EmployeeDAO definition
} // end namespace
```

Referring to Example 20.32 — I removed the `using` System.Drawing and `using` System.Drawing.Imaging directives, and made modifications to the InsertEmployee(), UpdateEmployee(), and FillInEmployeeVO() methods to properly handle the insertion and retrieval of a byte_array. Actually, a byte array was already being inserted and retrieved from the database. The only changes I made involved the elimination of the image conversion step. The code is actually simplified now that there's no need to convert an image into an array of bytes. However, this conversion will now need to be performed in the client application when an employee picture is selected for insertion.

Example 20.33 gives the code for the modified EmployeeGridClickedHandler() method which is found in the EmployeeTrainingClient class.

20.33 EmployeeGridClickedHandler() Method (modified)

```
1   private void EmployeeGridClickedHandler(Object sender, EventArgs e){
2       int selected_row = _employeeGrid.SelectedRows[ 0] .Index;
3       byte[ ] pictureBytes = _employeeList[ selected_row] .Picture;
4
5       if(pictureBytes !=  null){
6         MemoryStream ms = new MemoryStream();
7         ms.Write(pictureBytes, 0, pictureBytes.Length);
8         _pictureBox.Image = new Bitmap(ms);
9       } else {
10          _pictureBox.Image = null;
11        }
12      Console.WriteLine(selected_row);
13      Console.WriteLine(_employeeList[ selected_row] );
14
15      _trainingGrid.DataSource = null;
16      _trainingGrid.DataSource =
17               _employeeTraining.GetTrainingForEmployee(_employeeList[ selected_row] .EmployeeID);
18
19    }
```

Referring to Example 20.33 — the selected employee's Picture array is assigned to the pictureBytes reference. The `if` statement beginning on line 5 checks to see if the pictureBytes reference is not null. If it's not null, the pictureBytes array is written to a MemoryStream object, which is then used to create a Bitmap object.

Let's test these changes before proceeding further. You'll need to recompile the server application and copy the IEmployeeTraining.dll and VOLib.dll files to the client's ref folder. Start the server and then run the client. Figure 20-53 shows the client application with an employee's picture displayed in the PictureBox.

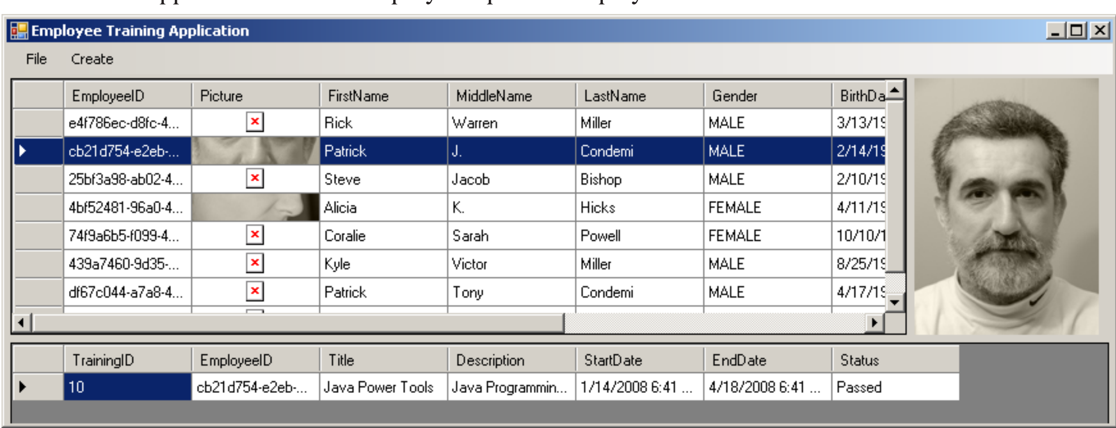


Figure 20-53: EmployeeTrainingClient Application with Employee's Picture Displayed in the PictureBox

Referring to Figure 20-53 — it seems the byte array is the way to go. You can also see a portion of each employee's picture (those that have one) in the corresponding cell under the Picture column. However, I'm not sure I want the employee picture in the DataGridView as it would make each row too high. I'll fix this in the next development iteration as well as add the ability to create and edit employees and their associated training.

Sixth Iteration

Now that an employee's data, including their image data, can be successfully transferred across the network, it's time to add more features to the EmployeeTrainingClient application. One thing I'll do will be to customize the DataGridViews and hide a few of the columns I don't want to display. I'll also add the ability to create, edit, and delete employees and training records. I'll use separate forms to enter and edit employee and training data. Table 20-10 lists the design considerations and design decisions for the sixth iteration.

Check-Off	Design Consideration	Design Decision
	Hide unwanted DataGridView columns.	The columns displayed in a DataGridView correspond to public properties of the EmployeeVO and TrainingVO classes. For the employee's DataGridView I'll hide the EmployeeID, Picture, FullName, and FullNameAndAge columns. For the training DataGridView I'll hide the EmployeeID and TrainingID columns.

	Application menus.	I think I'll do a redesign here and rename the Create menu and call it the Edit menu instead. To the Edit menu I'll add the following menu items:
		`Create Employee...`
		`Edit Employee`
		`---------`
		`Create Training...`
		`Edit Training...`
		`---------`
		`Delete Employee...`
		`Delete Training...`
		I'll need to do some menu manipulation while the application is running so I will move the declaration of the menu items out of the InitializeComponent() method so that I have access to them throughout the application. I'll also need to use a MessageBox to give users the chance to change their mind about deleting an employee or a training record.

	Employee form	I'll need to create a data entry form suitable for use both to create a new employee and to edit an existing employee. (Note: I could create and edit via the DataGridView but I'll leave that as an exercise for you!)
	Training form	I'll also need a data entry form suitable for use both to create and edit training records.

Table 20-10: Employee Training Client Application — Sixth Iteration Design Considerations And Decisions

I think I'll start by designing and implementing the data entry forms. Figure 20-54 shows the mock-up for the employee data entry form.

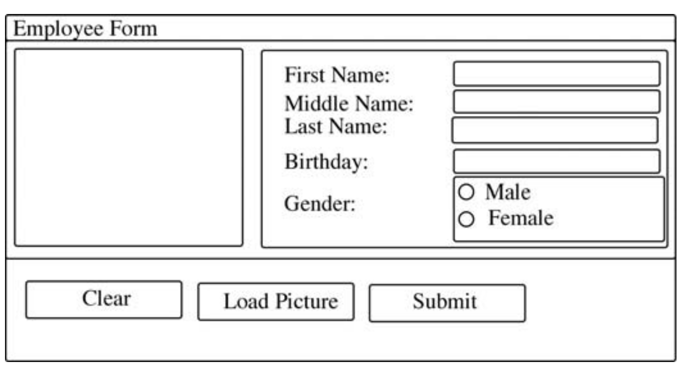


Figure 20-54: Employee Form Mock-up

Referring to Figure 20-54 — the employee form will contain the components required to enter and edit employee information. The components I'll need to use include Labels, TextBoxes, RadioButtons and a GroupBox, Buttons, and a PictureBox. I'll arrange the components with the help of several TableLayoutPanels and a FlowLayoutPanel.

I'll need a way to set and get the values of each data entry component. I'll make this possible by adding read-write properties to the employee form. Example 20.34 gives the code for the EmployeeForm class.

20.34 EmployeeForm.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;
using EmployeeTraining.VO;

public class EmployeeForm : Form {
  // constants
  private const int WINDOW_HEIGHT = 300;
  private const int WINDOW_WIDTH = 550;

  // fields
  private TableLayoutPanel _mainTablePanel;
  private TableLayoutPanel _infoTablePanel;
  private FlowLayoutPanel _buttonPanel;
  private PictureBox _pictureBox;
  private Label _firstNameLabel;
  private Label _middleNameLabel;
  private Label _lastNameLabel;
  private Label _birthdayLabel;
  private Label _genderLabel;
  private TextBox _firstNameTextBox;
  private TextBox _middleNameTextBox;
  private TextBox _lastNameTextBox;
  private DateTimePicker _birthdayPicker;
  private GroupBox _genderBox;
  private RadioButton _maleRadioButton;
  private RadioButton _femaleRadioButton;
  private Button _clearButton;
  private Button _loadPictureButton;
  private Button _submitButton;
  private OpenFileDialog _dialog;
  private bool _createMode;


  // public properties -
  public String FirstName {
    get { return _firstNameTextBox.Text; }
    set { _firstNameTextBox.Text = value; }
  }

  public String MiddleName {
    get { return _middleNameTextBox.Text; }
    set { _middleNameTextBox.Text = value; }
  }

  public String LastName {
    get { return _lastNameTextBox.Text; }
    set { _lastNameTextBox.Text = value; }
  }

  public DateTime Birthday {
    get { return _birthdayPicker.Value; }
    set { _birthdayPicker.Value = value; }
  }

  public Image Picture {
    get { return _pictureBox.Image; }
    set { _pictureBox.Image = value; }
  }

  public PersonVO.Sex Gender {
    get { return this.RadioButtonToSexEnum(); }
    set { this.SetRadioButton(value); }
  }

  public bool CreateMode {
    get { return _createMode; }
    set { _createMode = value; }
  }

  public bool SubmitOK {
    set { _submitButton.Enabled = value; }
```



```
  }

  public EmployeeForm(EmployeeTrainingClient externalHandler){
    this.InitializeComponent(externalHandler);
  }

  private void InitializeComponent(EmployeeTrainingClient externalHandler){
   _mainTablePanel = new TableLayoutPanel();
   _mainTablePanel.RowCount = 2;
   _mainTablePanel.ColumnCount = 2;
   _mainTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right
                 | AnchorStyles.Left;
   _mainTablePanel.Height = 500;
   _mainTablePanel.Width = 700;
   _infoTablePanel = new TableLayoutPanel();
   _infoTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right
                 | AnchorStyles.Left;
   _infoTablePanel.RowCount = 2;
   _infoTablePanel.ColumnCount = 2;
   _infoTablePanel.Height = 200;
   _infoTablePanel.Width = 400;
   _buttonPanel = new FlowLayoutPanel();
   _buttonPanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;
   _buttonPanel.Width = 500;
   _buttonPanel.Height = 200;

   _pictureBox = new PictureBox();
   _pictureBox.Height = 200;
   _pictureBox.Width = 200;
   _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Right | AnchorStyles.Left;

   _firstNameLabel = new Label();
   _firstNameLabel.Text = "First Name:";
   _middleNameLabel = new Label();
   _middleNameLabel.Text = "Middle Name:";
   _lastNameLabel = new Label();
   _lastNameLabel.Text = "Last Name:";
   _birthdayLabel = new Label();
   _birthdayLabel.Text = "Birthday";
   _genderLabel = new Label();
   _genderLabel.Text = "Gender";
   _firstNameTextBox = new TextBox();
   _firstNameTextBox.Width = 200;
   _middleNameTextBox = new TextBox();
   _middleNameTextBox.Width = 200;
   _lastNameTextBox = new TextBox();
   _lastNameTextBox.Width = 200;
   _birthdayPicker = new DateTimePicker();
   _genderBox = new GroupBox();
   _genderBox.Text = "Gender";
   _genderBox.Height = 75;
   _genderBox.Width = 200;
   _maleRadioButton = new RadioButton();
   _maleRadioButton.Text = "Male";
   _maleRadioButton.Checked = true;
   _maleRadioButton.Location = new Point(10, 20);
   _femaleRadioButton = new RadioButton();
   _femaleRadioButton.Text = "Female";
   _femaleRadioButton.Location = new Point(10, 40);
   _genderBox.Controls.Add(_maleRadioButton);
   _genderBox.Controls.Add(_femaleRadioButton);
   _clearButton = new Button();
   _clearButton.Text = "Clear";
   _clearButton.Click += this.ClearButtonHandler;
   _loadPictureButton = new Button();
   _loadPictureButton.Text = "Load Picture";
   _loadPictureButton.AutoSize = true;
   _loadPictureButton.Click += this.LoadPictureButtonHandler;
   _submitButton = new Button();
   _submitButton.Text = "Submit";
   _submitButton.Click += externalHandler.EmployeeSubmitButtonHandler;
   _submitButton.Enabled = false;

   _infoTablePanel.SuspendLayout();
   _infoTablePanel.Controls.Add(_firstNameLabel);
   _infoTablePanel.Controls.Add(_firstNameTextBox);
   _infoTablePanel.Controls.Add(_middleNameLabel);
   _infoTablePanel.Controls.Add(_middleNameTextBox);
   _infoTablePanel.Controls.Add(_lastNameLabel);
   _infoTablePanel.Controls.Add(_lastNameTextBox);
   _infoTablePanel.Controls.Add(_birthdayLabel);
```



```
   _infoTablePanel.Controls.Add(_birthdayPicker);
   _infoTablePanel.Controls.Add(_genderLabel);
   _infoTablePanel.Controls.Add(_genderBox);
   _infoTablePanel.Dock = DockStyle.Top;

   _buttonPanel.SuspendLayout();
   _buttonPanel.Controls.Add(_clearButton);
   _buttonPanel.Controls.Add(_loadPictureButton);
   _buttonPanel.Controls.Add(_submitButton);

   _mainTablePanel.SuspendLayout();
   _mainTablePanel.Controls.Add(_pictureBox);
   _mainTablePanel.Controls.Add(_infoTablePanel);
   _mainTablePanel.Controls.Add(_buttonPanel);
   _mainTablePanel.SetColumnSpan(_buttonPanel, 2);

    this.SuspendLayout();
    this.Controls.Add(_mainTablePanel);
    this.Width = WINDOW_WIDTH;
    this.Height = WINDOW_HEIGHT;
    this.Text = "Employee Form";
    _infoTablePanel.ResumeLayout();
    _buttonPanel.ResumeLayout();
    _mainTablePanel.ResumeLayout();
    this.ResumeLayout();
    _dialog = new OpenFileDialog();
   _dialog.FileOk += this.LoadPicture;
  }

  private void ClearButtonHandler(Object sender, EventArgs e){
    this.ClearFields();
    _submitButton.Enabled = false;
  }

  private void LoadPictureButtonHandler(Object sender, EventArgs e){
    _dialog.ShowDialog();
  }

  private void LoadPicture(Object sender, EventArgs e){
    String filename = _dialog.FileName;
    _pictureBox.Image = new Bitmap(filename);
    _submitButton.Enabled = true;
  }

  public void ClearFields(){
    _firstNameTextBox.Text = String.Empty;
    _middleNameTextBox.Text = String.Empty;
    _lastNameTextBox.Text = String.Empty;
    _maleRadioButton.Checked = true;
    _birthdayPicker.Value = DateTime.Now;
    _pictureBox.Image = null;
  }

  private PersonVO.Sex RadioButtonToSexEnum(){
    PersonVO.Sex gender = PersonVO.Sex.MALE;
    if(_maleRadioButton.Checked){
      gender = PersonVO.Sex.MALE;
    } else{
        gender = PersonVO.Sex.FEMALE;
      }
    return gender;
  }

  private void SetRadioButton(PersonVO.Sex gender){
    if(gender == PersonVO.Sex.MALE){
      _maleRadioButton.Checked = true;
    } else{
      _femaleRadioButton.Checked = true;
    }
  }

} // end class definition
```

Referring to Example 20.34 — most of the code is straightforward. The class contains several constants, fields, properties, and event handlers. The _submitButton.Click event is handled by the EmployeeTrainingClient.EmployeeSubmitButtonHandler() method. The _clearButton and _loadPictureButton Click events are handled by local event handlers.

Note that most of the properties consist of simple get and set statements, however, the Gender property's get and set call methods to perform the heavy lifting. The reason for this is that the radio button settings must be translated

into Person.Sex enumeration values and vice versa. The CreateMode property is used to indicate whether the form is used to create a new employee or edit an existing employee.

Figure 20-55 shows the mock-up for the training form.

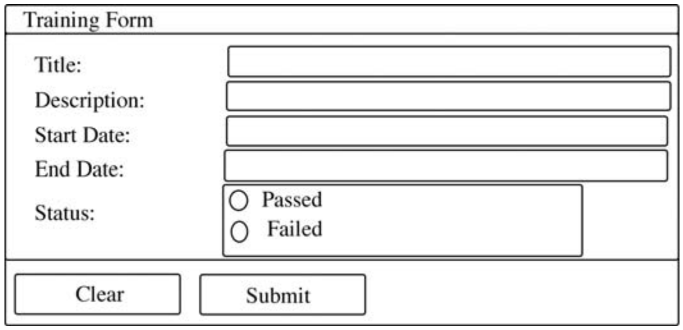


Figure 20-55: Training Form Mock-up

Referring to Figure 20-55 — the training form is built similar to the employee form. It will contain the data entry components required to create and edit an employee training record. It too uses TableLayoutPanels and a FlowLayoutPanel to arrange the components. Example 20.35 shows the code for the TrainingForm class.

20.35 TrainingForm .cs

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Collections.Generic;
using EmployeeTraining.VO;

public class TrainingForm : Form {
   // constants
   private const int WINDOW_HEIGHT = 300;
   private const int WINDOW_WIDTH = 450;
   private const bool DEBUG = true;

   // fields
   private TableLayoutPanel _mainTablePanel;
   private TableLayoutPanel _infoTablePanel;
   private FlowLayoutPanel _buttonPanel;
   private Label _titleLabel;
   private Label _descriptionLabel;
   private Label _startDateLabel;
   private Label _endDateLabel;
   private Label _statusLabel;
   private TextBox _titleTextBox;
   private TextBox _descriptionTextBox;
   private DateTimePicker _startDatePicker;
   private DateTimePicker _endDatePicker;
   private GroupBox _statusGroupBox;
   private RadioButton _passedRadioButton;
   private RadioButton _failedRadioButton;
   private Button _clearButton;
   private Button _submitButton;
   private bool _createMode;

   // public properties -
   public String Title {
     get {  return _titleTextBox.Text; }
     set {  _titleTextBox.Text = value; }
   }

   public String Description {
     get {  return _descriptionTextBox.Text; }
     set {  _descriptionTextBox.Text = value; }
   }

   public DateTime StartDate {
     get {  return _startDatePicker.Value; }
     set {  _startDatePicker.Value = value; }
   }

   public DateTime EndDate {
     get {  return _endDatePicker.Value; }
     set {  _endDatePicker.Value = value; }
```



```
    }

    public TrainingVO.TrainingStatus Status {
      get {  return this.RadioButtonToTrainingStatusEnum(); }
      set {  this.SetRadioButton(value); }
    }

    public bool CreateMode {
      get {  return _createMode; }
      set {  _createMode = value; }
    }

    public TrainingForm(EmployeeTrainingClient externalHandler){
      this.InitializeComponent(externalHandler);
    }

    private void InitializeComponent(EmployeeTrainingClient externalHandler){
      _mainTablePanel = new TableLayoutPanel();
      _mainTablePanel.RowCount = 2;
      _mainTablePanel.ColumnCount = 1;
      _mainTablePanel.Height = 400;
      _mainTablePanel.Width = 500;
      _mainTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
                  | AnchorStyles.Right;

      _infoTablePanel = new TableLayoutPanel();
      _infoTablePanel.RowCount = 5;
      _infoTablePanel.ColumnCount = 2;
      _infoTablePanel.Height = 200;
      _infoTablePanel.Width = 300;
      _infoTablePanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left
                    | AnchorStyles.Right;

      _buttonPanel = new FlowLayoutPanel();
      _buttonPanel.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;

      _titleLabel = new Label();
      _titleLabel.Text = "Title:";
      _descriptionLabel = new Label();
      _descriptionLabel.Text = "Description:";
      _startDateLabel = new Label();
      _startDateLabel.Text = "Start Date:";
      _endDateLabel = new Label();
      _endDateLabel.Text = "End Date";
      _statusLabel = new Label();
      _statusLabel.Text = "Status";
      _titleTextBox = new TextBox();
      _titleTextBox.Width = 300;
      _descriptionTextBox = new TextBox();
      _descriptionTextBox.Width = 300;
      _startDatePicker = new DateTimePicker();
      _endDatePicker = new DateTimePicker();
      _statusGroupBox = new GroupBox();
      _statusGroupBox.Height = 75;
      _statusGroupBox.Width = 300;
      _passedRadioButton = new RadioButton();
      _passedRadioButton.Text = "Passed";
      _passedRadioButton.Checked = true;
      _passedRadioButton.Location = new Point(10, 10);
      _failedRadioButton = new RadioButton();
      _failedRadioButton.Text = "Failed";
      _failedRadioButton.Location = new Point(10, 30);
      _clearButton = new Button();
      _clearButton.Text = "Clear";
      _clearButton.Click += this.ClearButtonHandler;
      _submitButton = new Button();
      _submitButton.Text = "Submit";
      _submitButton.Click += externalHandler.TrainingSubmitButtonHandler;

      _statusGroupBox.Controls.Add(_passedRadioButton);
      _statusGroupBox.Controls.Add(_failedRadioButton);

      _infoTablePanel.SuspendLayout();
      _infoTablePanel.Controls.Add(_titleLabel);
      _infoTablePanel.Controls.Add(_titleTextBox);
      _infoTablePanel.Controls.Add(_descriptionLabel);
      _infoTablePanel.Controls.Add(_descriptionTextBox);
      _infoTablePanel.Controls.Add(_startDateLabel);
      _infoTablePanel.Controls.Add(_startDatePicker);
      _infoTablePanel.Controls.Add(_endDateLabel);
      _infoTablePanel.Controls.Add(_endDatePicker);
```



```
      _infoTablePanel.Controls.Add(_statusLabel);
      _infoTablePanel.Controls.Add(_statusGroupBox);

      _buttonPanel.Controls.Add(_clearButton);
      _buttonPanel.Controls.Add(_submitButton);


      _mainTablePanel.SuspendLayout();
      _mainTablePanel.Controls.Add(_infoTablePanel);
      _mainTablePanel.Controls.Add(_buttonPanel);

      this.SuspendLayout();
      this.Controls.Add(_mainTablePanel);
      this.Height = WINDOW_HEIGHT;
      this.Width = WINDOW_WIDTH;
      this.Text = "Training Form";
      _infoTablePanel.ResumeLayout();
      _mainTablePanel.ResumeLayout();
      this.ResumeLayout();
    }

    private TrainingVO.TrainingStatus RadioButtonToTrainingStatusEnum(){
     TrainingVO.TrainingStatus status = TrainingVO.TrainingStatus.Passed;
     if(_passedRadioButton.Checked){
       status = TrainingVO.TrainingStatus.Passed;
     } else{
         status = TrainingVO.TrainingStatus.Failed;
     }
     return status;
   }

    private void ClearButtonHandler(Object sender, EventArgs e){
      this.ClearFields();
    }

    public void ClearFields(){
      _titleTextBox.Text = String.Empty;
      _descriptionTextBox.Text = String.Empty;
      _startDatePicker.Value = DateTime.Now;
      _endDatePicker.Value = DateTime.Now;
      _passedRadioButton.Checked = true;
    }

    private void SetRadioButton(TrainingVO.TrainingStatus status){
     if(status == TrainingVO.TrainingStatus.Passed){
       _passedRadioButton.Checked = true;
     } else{
       _failedRadioButton.Checked = true;
     }
   }
} // end class definition
```

Example 20.36 gives the code for the revised EmployeeTrainingClient class.

20.36 EmployeeTrainingClient.cs (revised)

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.ComponentModel;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using EmployeeTraining.VO;

public class EmployeeTrainingClient : Form {

  // Constants
  private const int WINDOW_HEIGHT = 500;
  private const int WINDOW_WIDTH = 900;
  private const String WINDOW_TITLE = "Employee Training Application";
  private const bool DEBUG = true;

  // fields
  private MenuStrip _ms;
  private ToolStripMenuItem _fileMenu;
  private ToolStripMenuItem _exitMenuItem;
  private ToolStripMenuItem _editMenu;
  private ToolStripMenuItem _createEmployeeMenuItem;
  private ToolStripMenuItem _createTrainingMenuItem;
```



```
    private ToolStripMenuItem _editEmployeeMenuItem;
    private ToolStripMenuItem _editTrainingMenuItem;
    private ToolStripMenuItem _deleteEmployeeMenuItem;
    private ToolStripMenuItem _deleteTrainingMenuItem;
    private IEmployeeTraining _employeeTraining = null;
    private List<EmployeeVO> _employeeList = null;
    private List<TrainingVO> _trainingList = null;
    private TableLayoutPanel _tablePanel = null;
    private DataGridView _employeeGrid = null;
    private DataGridView _trainingGrid = null;
    private PictureBox _pictureBox = null;
    private EmployeeForm _employeeForm;
    private TrainingForm _trainingForm;

    public EmployeeTrainingClient(IEmployeeTraining employeeTraining){
      _employeeTraining = employeeTraining;
      this.InitializeComponent();
    }

    private void InitializeComponent(){
      // setup the menus
      _ms = new MenuStrip();

      _fileMenu = new ToolStripMenuItem("File");
      _exitMenuItem = new ToolStripMenuItem("Exit", null, new EventHandler(this.ExitProgramHandler));

      _editMenu = new ToolStripMenuItem("Edit");
      _createEmployeeMenuItem = new ToolStripMenuItem("Create Employee...", null,
                       new EventHandler(this.CreateEmployeeHandler));
      _createTrainingMenuItem = new ToolStripMenuItem("Create Training...", null,
                       new EventHandler(this.CreateTrainingHandler));
      _editEmployeeMenuItem = new ToolStripMenuItem("Edit Employee...", null,
                       new EventHandler(this.EditEmployeeHandler));
      _editEmployeeMenuItem.Enabled = false;
      _editTrainingMenuItem = new ToolStripMenuItem("Edit Training...", null,
                     new EventHandler(this.EditTrainingHandler));
      _editTrainingMenuItem.Enabled = false;
      _deleteEmployeeMenuItem = new ToolStripMenuItem("Delete Employee...", null,
                       new EventHandler(this.DeleteEmployeeHandler));
      _deleteEmployeeMenuItem.Enabled = false;
      _deleteTrainingMenuItem = new ToolStripMenuItem("Delete Training...", null,
                       new EventHandler(this.DeleteTrainingHandler));
      _deleteTrainingMenuItem.Enabled = false;

      _fileMenu.DropDownItems.Add(_exitMenuItem);
      _ms.Items.Add(_fileMenu);

      _editMenu.DropDownItems.Add(_createEmployeeMenuItem);
      _editMenu.DropDownItems.Add(_createTrainingMenuItem);
      _editMenu.DropDownItems.Add("-");
      _editMenu.DropDownItems.Add(_editEmployeeMenuItem);
      _editMenu.DropDownItems.Add(_editTrainingMenuItem);
      _editMenu.DropDownItems.Add("-");
      _editMenu.DropDownItems.Add(_deleteEmployeeMenuItem);
      _editMenu.DropDownItems.Add(_deleteTrainingMenuItem);
      _ms.Items.Add(_editMenu);

      // create the table panel
      _tablePanel = new TableLayoutPanel();
      _tablePanel.RowCount = 2;
      _tablePanel.ColumnCount = 2;
      _tablePanel.Anchor =  AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
      _tablePanel.Dock = DockStyle.Top;
      _tablePanel.Height = 400;

      // create and initialize the data grids
      _employeeGrid = new DataGridView();
      _employeeGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
      _employeeGrid.Height = 200;
      _employeeGrid.Width = 700;
      _employeeList = _employeeTraining.GetAllEmployees();
      _employeeGrid.DataSource = _employeeList;
      _employeeGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
      _employeeGrid.Click += this.EmployeeGridClickedHandler;
      _employeeGrid.DataBindingComplete += this.EmployeeGridDataBindingCompleteHandler;

      _trainingGrid = new DataGridView();
      _trainingGrid.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
      _trainingGrid.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;
      _trainingGrid.DataBindingComplete += this.TrainingGridDataBindingCompleteHandler;

```



```

    _trainingList = _employeeTraining.GetTrainingForEmployee(_employeeList[0].EmployeeID);
    _trainingGrid.DataSource = _trainingList;

    // create picture box
    _pictureBox = new PictureBox();
    _pictureBox.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left | AnchorStyles.Right;


    //add grids to table panel
    _tablePanel.Controls.Add(_employeeGrid);
    _tablePanel.Controls.Add(_pictureBox);
    _tablePanel.Controls.Add(_trainingGrid);
    _tablePanel.SetColumnSpan(_trainingGrid, 2);

    this.Controls.Add(_tablePanel);
    _ms.Dock = DockStyle.Top;
    this.MainMenuStrip = _ms;
    this.Controls.Add(_ms);
    this.Height = WINDOW_HEIGHT;
    this.Width = WINDOW_WIDTH;
    this.Text = WINDOW_TITLE;
    _employeeForm = new EmployeeForm(this);
    _employeeForm.Visible = false;
    _trainingForm = new TrainingForm(this);
    _trainingForm.Visible = false;
  }

  /**********************************************************
    Event Handlers
  **********************************************************/
  private void ExitProgramHandler(Object sender, EventArgs e){
    Application.Exit();
  }

  private void CreateEmployeeHandler(Object sender, EventArgs e){
    _employeeForm.CreateMode = true;
    _employeeForm.SubmitOK = false;
    _employeeForm.ClearFields();
    _employeeForm.ShowDialog();
  }

  private void CreateTrainingHandler(Object sender, EventArgs e){
    _trainingForm.CreateMode = true;
    _trainingForm.ClearFields();
    _trainingForm.ShowDialog();
  }

  private void EditEmployeeHandler(Object sender, EventArgs e){
    _employeeForm.ClearFields();
    _employeeForm.SubmitOK = true;
    _employeeForm.CreateMode = false;
    EmployeeVO vo = _employeeList[_employeeGrid.SelectedRows[0].Index];
    _employeeForm.FirstName = vo.FirstName;
    _employeeForm.MiddleName = vo.MiddleName;
    _employeeForm.LastName = vo.LastName;
    _employeeForm.Birthday = vo.BirthDay;
    _employeeForm.Gender = vo.Gender;
    MemoryStream ms = new MemoryStream();
    if(vo.Picture != null) {
      ms.Write(vo.Picture, 0, vo.Picture.Length);
      _employeeForm.Picture = new Bitmap(ms);
    }
    _employeeForm.ShowDialog();
  }

  private void EditTrainingHandler(Object sender, EventArgs e){
    _trainingForm.CreateMode = false;
    TrainingVO vo = _trainingList[_trainingGrid.SelectedRows[0].Index];
    _trainingForm.Title = vo.Title;
    _trainingForm.Description = vo.Description;
    _trainingForm.StartDate = vo.StartDate;
    _trainingForm.EndDate = vo.EndDate;
    _trainingForm.Status = vo.Status;
    _trainingForm.ShowDialog();
  }

  private void EmployeeGridClickedHandler(Object sender, EventArgs e){
    int selected_row = _employeeGrid.SelectedRows[0].Index;
    byte[] pictureBytes = _employeeList[selected_row].Picture;

```



```
    if(pictureBytes !=  null){
      MemoryStream ms = new MemoryStream();
      ms.Write(pictureBytes, 0, pictureBytes.Length);
      _pictureBox.Image = new Bitmap(ms);
    } else {
        _pictureBox.Image = null;
      }
    Console.WriteLine(selected_row);
    Console.WriteLine(_employeeList[selected_row]);

    _trainingGrid.DataSource = null;
    _trainingList = _employeeTraining.GetTrainingForEmployee(_employeeList[selected_row].EmployeeID);
    _trainingGrid.DataSource = _trainingList;
    if(_trainingList.Count > 0){
      _trainingGrid.Rows[0].Selected = true;
      _editTrainingMenuItem.Enabled = true;
      _deleteTrainingMenuItem.Enabled = true;
    } else {
      _editTrainingMenuItem.Enabled = false;
      _deleteTrainingMenuItem.Enabled = false;
    }

    if(DEBUG){
      foreach(EmployeeVO emp in _employeeList){
        Console.WriteLine(emp.FirstName + " " + emp.LastName);
      }
    }
  }

  private void EmployeeGridDataBindingCompleteHandler(Object sender, EventArgs e){
    _employeeGrid.Columns["Picture"].Visible = false;
    _employeeGrid.Columns["FullName"].Visible = false;
    _employeeGrid.Columns["FullNameAndAge"].Visible = false;
    _employeeGrid.Columns["Age"].ReadOnly = true;
    _employeeGrid.Columns["Age"].ToolTipText = "Read Only!";
    _employeeGrid.Columns["EmployeeID"].Visible = false;
    if(_employeeList.Count > 0){
      _employeeGrid.Rows[0].Selected = true;
      this.EmployeeGridClickedHandler(this, new EventArgs());
      _editEmployeeMenuItem.Enabled = true;
      _deleteEmployeeMenuItem.Enabled = true;
    }
  }

  private void TrainingGridDataBindingCompleteHandler(Object sender, EventArgs e){
    _trainingGrid.Columns["TrainingID"].Visible = false;
    _trainingGrid.Columns["EmployeeID"].Visible = false;
    if(_trainingList.Count > 0){
      _trainingGrid.Rows[0].Selected = true;
      _editTrainingMenuItem.Enabled = true;
      _deleteTrainingMenuItem.Enabled = true;
    }
  }

  public void EmployeeSubmitButtonHandler(Object sender, EventArgs e){
    if(_employeeForm.CreateMode){  // creating new employee
      EmployeeVO vo = new EmployeeVO();
      vo.FirstName = _employeeForm.FirstName;
      vo.MiddleName = _employeeForm.MiddleName;
      vo.LastName = _employeeForm.LastName;
      vo.BirthDay = _employeeForm.Birthday;
      MemoryStream ms = new MemoryStream();
      _employeeForm.Picture.Save(ms, ImageFormat.Tiff);
      vo.Picture = ms.ToArray();
      vo.Gender = _employeeForm.Gender;
      _employeeTraining.CreateEmployee(vo);
      _employeeForm.Visible = false;
      _employeeList = _employeeTraining.GetAllEmployees();
      _employeeGrid.DataSource = _employeeList;
      _employeeForm.ClearFields();
    }else{  // editing new employee
       EmployeeVO vo = _employeeList[_employeeGrid.SelectedRows[0].Index];
       vo.FirstName = _employeeForm.FirstName;
       vo.MiddleName = _employeeForm.MiddleName;
       vo.LastName = _employeeForm.LastName;
       vo.BirthDay = _employeeForm.Birthday;
       MemoryStream ms = new MemoryStream();
       _employeeForm.Picture.Save(ms, ImageFormat.Tiff);
       vo.Picture = ms.ToArray();
       vo.Gender = _employeeForm.Gender;
       _employeeTraining.UpdateEmployee(vo);
```



```
        _employeeForm.Visible = false;
        _employeeList = _employeeTraining.GetAllEmployees();
        _employeeGrid.DataSource = _employeeList;
        _employeeForm.ClearFields();

     }
   }

   public void TrainingSubmitButtonHandler(Object sender, EventArgs e){
     if(_trainingForm.CreateMode){
       TrainingVO vo = new TrainingVO();
       int selected_row = _employeeGrid.SelectedRows[0].Index;
       vo.EmployeeID = _employeeList[selected_row].EmployeeID;
       vo.Title = _trainingForm.Title;
       vo.Description = _trainingForm.Description;
       vo.StartDate = _trainingForm.StartDate;
       vo.EndDate = _trainingForm.EndDate;
       vo.Status = _trainingForm.Status;
       _employeeTraining.CreateTraining(vo);
       _trainingGrid.DataSource = null;
       _trainingGrid.DataSource = _employeeTraining.GetTrainingForEmployee(vo.EmployeeID);
       _trainingForm.Visible = false;
       _trainingForm.ClearFields();
     } else {
       TrainingVO vo = _trainingList[_trainingGrid.Rows[0].Index];
       vo.Title = _trainingForm.Title;
       vo.Description = _trainingForm.Description;
       vo.StartDate = _trainingForm.StartDate;
       vo.EndDate = _trainingForm.EndDate;
       vo.Status = _trainingForm.Status;
       _employeeTraining.UpdateTraining(vo);
       _trainingGrid.DataSource = null;
       _trainingGrid.DataSource = _employeeTraining.GetTrainingForEmployee(vo.EmployeeID);
       _trainingForm.Visible = false;
       _trainingForm.ClearFields();
     }
   }

   private void DeleteEmployeeHandler(Object sender, EventArgs e){
     DialogResult result = MessageBox.Show("Are you sure? Click OK to delete, “ +
                           “or Cancel to return to the application.",
                                           "Warning!", MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);
     if(result == DialogResult.OK){
       int selected_row = _employeeGrid.SelectedRows[0].Index;
       _employeeTraining.DeleteEmployee(_employeeList[selected_row].EmployeeID);
       _employeeGrid.DataSource = null;
       _employeeList = _employeeTraining.GetAllEmployees();
       _employeeGrid.DataSource = _employeeList;
       if(_employeeList.Count > 0){
         _employeeGrid.Rows[0].Selected = true;
         this.EmployeeGridClickedHandler(this, new EventArgs());
         _editEmployeeMenuItem.Enabled = true;
         _deleteEmployeeMenuItem.Enabled = true;
       }
     }
   }

   private void DeleteTrainingHandler(Object sender, EventArgs e){
      DialogResult result = MessageBox.Show("Are you sure? Click OK to delete, “ +
                            “or Cancel to return to the application.",
                                            "Warning!", MessageBoxButtons.OKCancel, MessageBoxIcon.Warning);
      if(result == DialogResult.OK){
        int selected_row = _trainingGrid.SelectedRows[0].Index;
        _employeeTraining.DeleteTraining(_trainingList[selected_row].TrainingID);
        _trainingGrid.DataSource = null;
        int selected_employee = _employeeGrid.SelectedRows[0].Index;
        _trainingList =
            _employeeTraining.GetTrainingForEmployee(_employeeList[selected_employee].EmployeeID);
        _trainingGrid.DataSource = _trainingList;
        if(_trainingList.Count > 0){
          _trainingGrid.Rows[0].Selected = true;
          _editTrainingMenuItem.Enabled = true;
          _deleteTrainingMenuItem.Enabled = true;
        }
      }
   }

   [STAThread]
   public static void Main(){
    try {
      RemotingConfiguration.Configure("EmployeeTrainingClient.exe.config", false);
```

```
352       WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
353       IEmployeeTraining employee_training =
354         (IEmployeeTraining)Activator.GetObject(typeof(IEmployeeTraining), client_types[0].ObjectUrl );
355       EmployeeTrainingClient client = new EmployeeTrainingClient(employee_training);
356       Application.Run(client);
357     }catch(Exception e){
358       Console.WriteLine(e);
359     }
360   }
361 } // end class definition
```

Referring to Example 20.36 — well, there's a lot going on here but it should be easy to follow the code. First, I've moved the declarations for the menu and its menu items into the fields area so I can have access to menu items when I need to manipulate them. "What will I be doing?" you ask. Well, for one thing, I want to disable the "Edit Employee..." and "Delete Employee..." menu choices when there are no employees to edit or delete. I also want to do the same for the "Edit Training..." and "Delete Training..." menu choices.

I would like to focus your attention on a few areas of the code worth special mention. First, before I can hide any columns, I must wait until the DataGridView controls have been properly data bound. Data binding takes place when I assign a data source to a DataGridView's DataSource property. When data binding is complete the control fires the DataBindingComplete event. The column-hiding code for the _employeeGrid is placed in the EmployeeGridDataBindingCompleteHandler() method, which begins on line 219. To get an employee's picture to load into the _pictureBox and their associated training records to display in the _trainingGrid, I make an explicit call to the EmployeeGridClickedHandler() method on line 228.

The _trainingGrid.DataBindingComplete event is handled by the TrainingGridDataBindingCompleteHandler() method which begins on line 234. I place the column-hiding code for the _trainingGrid in this method.

Compiling And Running The Modified EmployeeTrainingClient Project

I placed the EmployeeForm.cs and TrainingForm.cs files in the project's app directory. To compile these files along with the EmployeeTrainingClient.cs file, I need to make a minor change to the EmployeeTrainingClient.proj file. Example 20.37 gives the modified project file.

20.37 EmployeeTrainingClient.proj (modified)

```
<Project DefaultTargets="Run"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

   <PropertyGroup>
     <IncludeDebugInformation>false</IncludeDebugInformation>
     <BuildDir>build</BuildDir>
     <AppDir>app</AppDir>
     <RefDir>ref</RefDir>
     <ConfigDir>config</ConfigDir>
   </PropertyGroup>

    <ItemGroup>

      <APP Include="app\**\*.cs" />
      <REF Include="ref\**\*.dll" />
      <CONFIG Include="config\**\*.config" />
      <EXE Include="app\**\*.exe" />
    </ItemGroup>

    <Target Name="MakeDirs">
      <MakeDir Directories="$(BuildDir)" />
    </Target>

    <Target Name="RemoveDirs">
      <RemoveDir Directories="$(BuildDir)" />
    </Target>

    <Target Name="Clean"
            DependsOnTargets="RemoveDirs;MakeDirs">
    </Target>

     <Target Name="CopyFiles">
       <Copy
         SourceFiles="@(CONFIG);@(REF)"
         DestinationFolder="$(BuildDir)" />
    </Target>

    <Target Name="CompileApp"
            Inputs="@(APP)"
```



```
40              Outputs="$(BuildDir)\$(MSBuildProjectName).exe"
41              DependsOnTargets="Clean">
42        <Csc Sources="@(APP)"
43             TargetType="exe"
44             References="@(REF)"
45             OutputAssembly="$(BuildDir)\$(MSBuildProjectName).exe">
46        </Csc>
47      </Target>
48
49      <Target Name="Run"
50              DependsOnTargets="CompileApp;CopyFiles">
51        <Exec Command="$(MSBuildProjectName).exe"
52              WorkingDirectory="$(BuildDir)" />
53      </Target>
54  </Project>
```

Referring to Example 20.37 — the change appears on line 14 where I've specified the <APP> item to include all the source files found in the project's app folder.

To compile and run the EmployeeTrainingClient project, make sure the server is up and running, change to the EmployeeTrainingClient project directory, and enter the following command-line command:

```
msbuild
```

This executes the default build target. If all goes well you'll see the application window appear. Figure 20-56 shows the main application window with the Edit menu extended to show the new menu items.

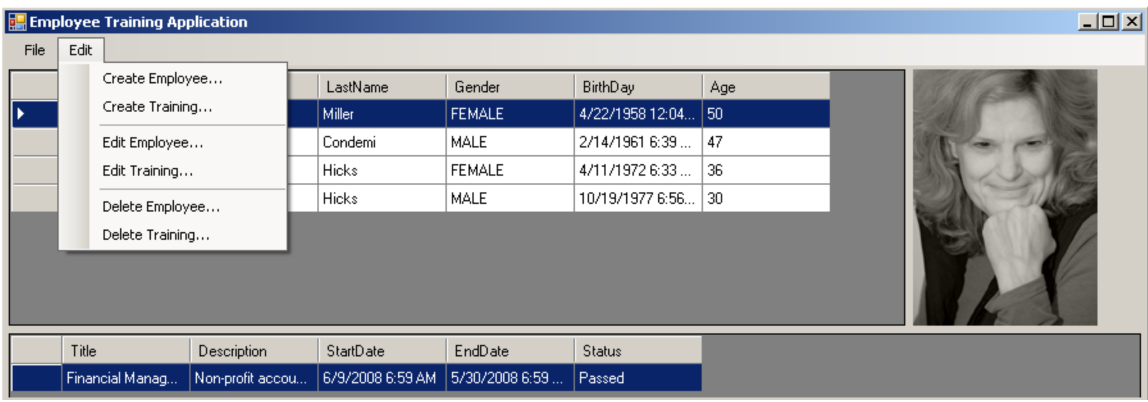


Figure 20-56: Main Application Window with Edit Menu Open to Reveal Revised Menu Structure

Figure 20-57 shows how the Edit menu looks when some of the menu items are disabled.

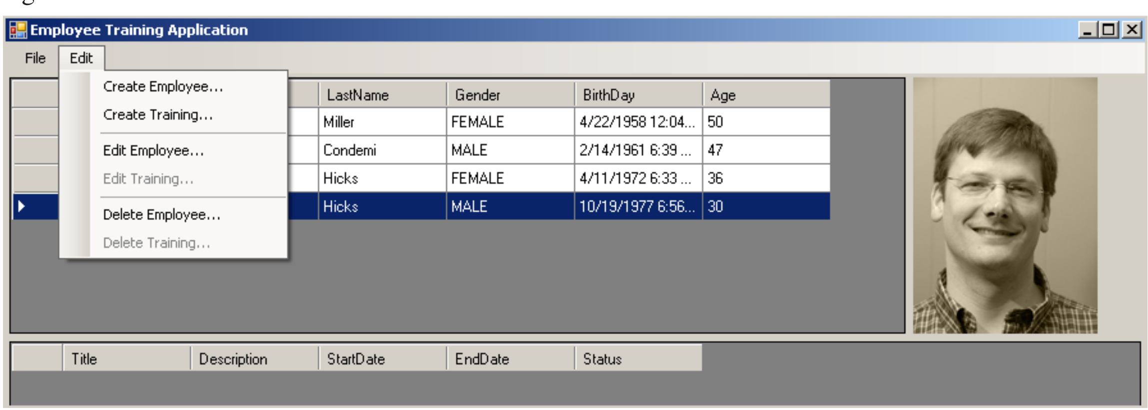


Figure 20-57: Edit Menu Items Disabled

Referring to Figure 20-57 — Bill Hicks has no training so the "Edit Training..." and "Delete Training..." menu items are disabled.

To create a new employee select Edit->Create Employee... to open the employee form, as is shown in Figure 20-58. Referring to Figure 20-58 — the employee form is cleared when creating a new employee and its Submit button is disabled. To enable the Submit button you need to load a picture. Figure 20-59 shows how the employee form looks fully populated. Figure 20-60 shows how the training form looks empty and fully populated.

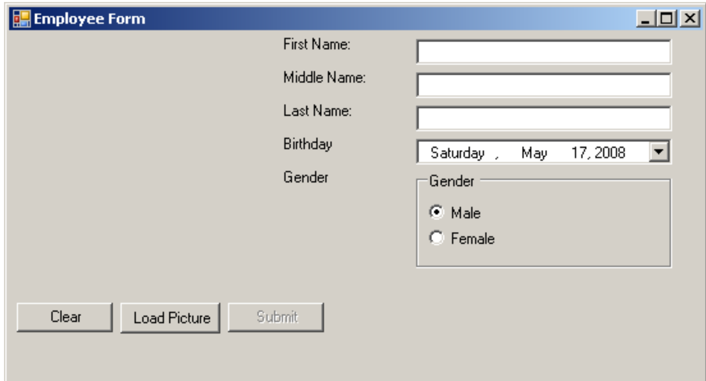


Figure 20-58: Empty Employee Data Entry Form

Figure 20-59: Employee Form Fully Populate and Submit Button Enabled

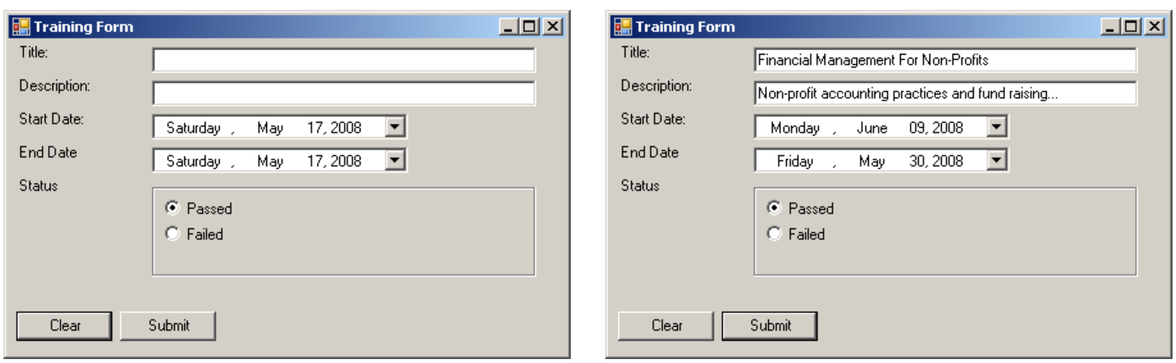


Figure 20-60: Training Form Empty and Filled

Where To Go From Here

The DataGridView control is extremely powerful and in the EmployeeTrainingClient application I don't come close to tapping its full potential. So, for starters, I recommend you explore its capabilities further by spending some time on MSDN and researching its members. For example, it's not necessary to have separate data entry forms to enter and edit employee and training data. You can create new DataGridView rows programmatically and edits made to data contained therein are reflected in the bound data source. I've put some code in the EmployeeTrainingClient application that shows how the EmployeeVO objects contained in the _employeeList are changed automatically when you edit an _employeeGrid column. (See Example 20.36 lines 212 - 216)

Regarding the database side of things, although I covered a lot of ground, I omitted topics such as normal forms and mapping tables. I'll leave you to explore these and other database topics on your own. Having seen the employee training project developed from start to finish should have filled your head with so many ideas that they are falling out of your ears!

Summary

Relational databases hold data in *tables*. Table *columns* are specified to be of a particular *data type*. Table data is contained in *rows*. Structured Query Language (SQL) is used to *create*, *manipulate*, and *delete* relational database objects and data. SQL contains three sub-languages: Data Definition Language (DDL) which is used to create databases, tables, views, and other database objects; Data Manipulation Language (DML) which is used to create, manipulate, and delete the data contained within a database; and Data Control Language (DCL) which is used to grant or revoke user rights and privileges on database objects.

Different database makers are free to extend SQL to suit their needs so there's no guarantee of SQL portability between different databases.

One or more table columns can be designated as a *primary key* whose value is unique for each row inserted into that table. Related tables can be created by including the primary key of one table as a *foreign key* in the related table.

The select command can be used to construct complex queries involving multiple related tables. One table is joined to another to form a temporary table. There are many different types of *join* operations, but the most common one is an inner join, which is the default join condition provided by Microsoft SQL Server.

Inner joins are made possible through the use of foreign keys. A *foreign key* is a column in a table that contains a value that is used as a primary key in another table. A table can be related to many other tables by including multiple foreign keys. Specify a foreign key by adding a foreign key constraint to a particular table using the alter command.

Use *database scripts* to ease database development. Scripts that create the database, tables, constraints, and test data let you work at the speed of light.

Approach the design and implementation of complex database applications in an iterative fashion. Structure the design of your application in such a way as to make changing the application as painless as possible. A tiered approach to application design allows you to quickly identify and correct problems or make application modifications when you realize your design needs to be changed.

Transfer complex data types as byte arrays (byte[]) and convert them into the appropriate type at the other end.

References

Microsoft Patterns and Practices Developer Center. [http://www.codeplex.com/entlib]

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Candace C. Fleming & Barbara von Halle. Handbook of Relational Database Design, Addison-Wesley Professional, 1989, ISBN: 0-201-11434-8

Notes

Appendices

Appendix A

Numeric String Formatting

Numeric Formatting

C# makes it easy to format numeric strings. You can format numeric results using the String.Format() method or the Console.Write() or Console.WriteLine() methods.

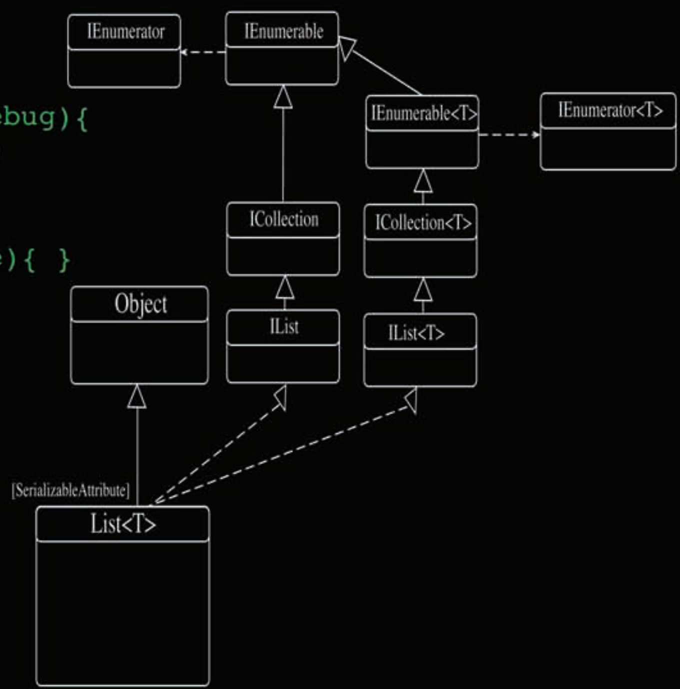
A format string takes the form C_f*nn* where C_f is a format specifier character and *nn* specifies the number of decimal digits. Table Appendix A-1 lists the standard C# numeric format strings along with some brief example code.

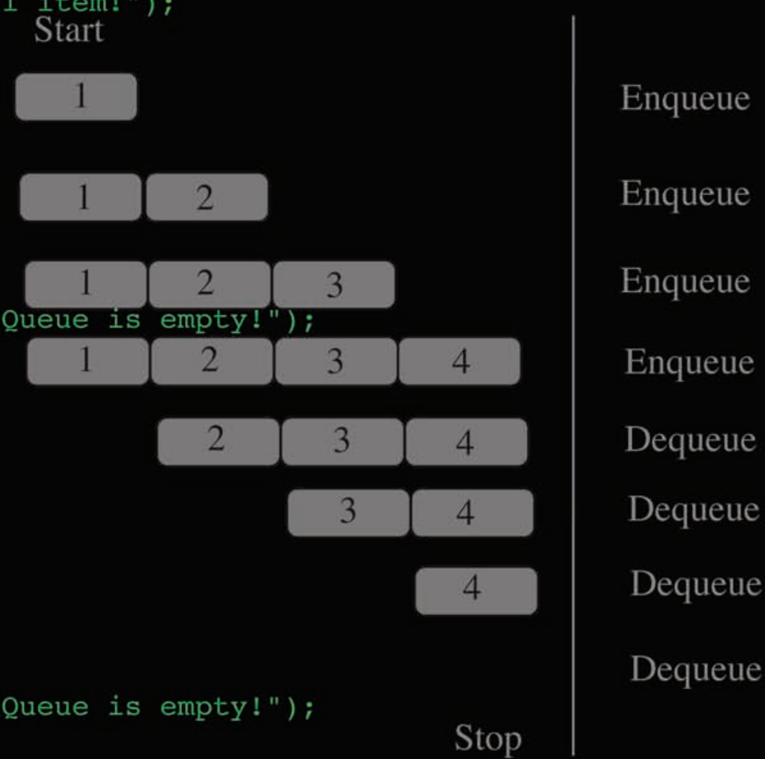
Character	Description	Example Code	Results
C or c	Currency	Console.Write("{0:C}", 4.5); Console.Write("{0:C}", -4.5);	$4.50 ($4.50)
D or d	Decimal	Console.Write("{0:D5}", 45);	00045
E or e	Scientific	Console.Write("{0:E}", 450000);	4.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 45); Console.Write("{0:F0}", 45);	45.00 45
G or g	General	Console.Write("{0:G}", 4.5);	4.5
N or n	Number	Console.Write("{0:N}", 4500000);	4,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 450); Console.Write("{0:X}", 0xabcd);	1C2 ABCD

Table Appendix A -1: Numeric Formatting

Index


```
using System;

public class HomeGrownQueue {
  private CircularArray _ca = null;
  private const int INITIAL_SIZE = 25;

  public HomeGrownQueue(int initial_size, bool debug){
    _ca = new CircularArray(initial_size, debug);
  }

  public HomeGrownQueue():this(INITIAL_SIZE, true){ }

  public bool IsEmpty {
    get { return _ca.IsEmpty; }
  }

  public int Count {
    get { return _ca.Count; }
  }

  public void Enqueue(object item){
    try{
      _ca.Insert(item);
    }catch(Exception){
      Console.WriteLine("Cannot enqueue null item!");
    }
  }

  public object Dequeue(){
    object return_object = null;
    try{
     return_object = _ca.Remove();
    }catch(Exception){
      throw new InvalidOperationException("Queue is empty!");
    }
    return return_object;

  }

  public object Peek(){
    object return_object;
    try{
     return_object = _ca.Peek();
    }catch(Exception){
      throw new InvalidOperationException("Queue is empty!");
    }
    return return_object;
  }

}
```





Pulp Free Press
Falls Church, Virginia
Good for the brain!

ISBN-13: 9781932504095
55995
9 781932 504095